コンピュータ・サイエンス研究書シリーズ ㉒

タイムシェアリング・システム

カリフォルニア大学 R.W.ワトソン著
慶応義塾大学教授・工博 相磯秀夫訳

日本コンピュータ協会

コンピュータ・サイエンス研究書シリーズ

タイムシェアリング・システム

Richard W. Watson著
慶応義塾大学教授・工博、相磯秀夫訳

日本コンピュータ協会

Timesharing System Design Concepts

Richard W. Watson
Department of Electrical Engineering
and Computer Science
University of California, Berkeley

まえがき

タイムシェアリング・システムの分野は，多くの一般的な設計目標ならびに一般的な設計や実現に関する概念および機構が種々のシステムで使われていると認識できる点にまで成長してきた．本書における目標はタイムシェアリング・システム設計の中核になってきたと思われる一般的な概念や技術を選び出して述べることである．実例を通していくつかの概念を示し，またそれが開発されたいきさつと基本的な機構を紹介していく．このような方法を採る理由は，私自身や他の人達の注意深い推察から，一度ある特定の関係で理解した考え方は他のものにおいても容易に理解でき，一般化でき，しかも拡張できるという経験をしているからである．一般にこれはプログラミングを学習する際にとる手段である．すなわち，1つの言語の文脈において識別子，サブルーチン，ループおよび条件文などの基本的概念を学ぶと，その他の言語でこれらの概念がどのように取扱われているかを容易に理解したり批評することができる．

読者が概念に焦点を合わせやすいように，その概念のいくつかの可能な実現方法を詳説することはせず，2つのシステムから多くの実例をあげるようにした．そのシステムは California 大学 Berkeley で開発された XDS-940 タイムシェアリング・システム* と MIT で開発された GE-645 Multics タイムシェアリング・システム† である．有用と思われるときにはその他のシステムからも実例を引用した．各々の例は考察中の概念を理解したり，その概念が解決あるいは作り出す問題点を理解する上で必要なときのみ詳細な点まで述べることにした．

次の理由により XDS-940 と Multics システムを選んだ．(1) 初期のシステムで得た多くの経験を包含していること，(2) この分野での概念の発展に重要な影響を及ぼしていること，そして，(3) この2つのシステムは設計方針で重要な差異があることである．XDS-940 を選んだ別の理由は私が身近に接した

---

* W. W. Lichtenberger and M. W. Pirtle, A Facility for Experimentation in Man-Machine Interaction, *AFIPS Conf. Proc. Fall Joint Computer Conf.*, vol. 27, pp. 589-598, 1965.

† F. J. Corbato and V. A. Vyssotsky, Introduction and Overview of the Multics System, *AFIPS Conf. Proc. Fall Joint Computer Cont.*, vol. 27 : pp. 185-196. 1965.

システムであったからである．これは中規模のシステムである．XDS-940 の設計開発者の目標は高度な会話形式で限られたユーザ数を取扱う非常に融通性のあるシステムを開発することであった．大部分のユーザ・プログラムは 16,000 語以下の大きさである．California 大学で開発されたシステムの概念はさらに発展され，企業や政府の研究所で XDS-940 の商用機として成功を導びいた．さらに重要なことは商用タイムシェアリング・サービス会社で広く使われるようになったことである．Multics システムは大規模なシステムである．その研究設計者の目標は広範囲の問題分野で仕事をしている大ユーザ社会の計算サービスを行うシステムを開発することであった．ユーザに大型計算機と仮想記憶を提供し，高度な一般性をもたせて，共用するプログラムやデータに対してただ1つの複写を提供するという2つの特別な目標が設定された．このプロジェクトは概念的な開発で大きな成果をあげた．Multics で開発された概念は XDS-940 の開発にも利用され，また IBM 360/67 システムなど他のシステムにも強い影響を与えている．System Development Corporation タイムシェアリング・システム*，MIT の CTSS システム†，および Dartmouth システム††は本書では議論されていない．その理由はそこで開発された多くの概念は，本書の目的のためには XDS-940 および Multics システムでの例で示すことができるからである．

再度述べると，私の目標は基本的な概念や機構を包括することであり，本書に載せられている事柄を理解すれば，学生はさらに詳細な事や種々の実現されたものについての文献を容易に理解できると思う．同じように専門的な分野においても，本書を理解した後は設置するシステムあるいは設置計画のあるシステムのマニュアル類を一層容易に理解できるようになる．

本書は主に2つのグループのために書かれている：

1. 責任があって，あるいは興味からタイムシェアリング・システムで開発された事柄や基本的な原理を理解する必要があるが，そのようなシス

* J. Schwartz, E. G. Coffman, and L. Weissman, A General Purpose Time-sharing System, *AFIPS Conf. Proc. Spring Joint Computer Conf.*, vol. 25, pp. 397-411, 1964.

† F. J. Corbato et al., An Experimental Time-sharing System, *AFIPS Conf. Proc. Spring Joint Computer Conf.*, vol. 21, pp. 335-344, 1962.

†† R. F. Hargraves, Jr., and A. G. Stephenson, Design Considerations for an Educational Time-sharing System, *AFIPS Conf. Proc. Spring Joint Computer Conf.*, vol. 34, pp. 657-664, 1969

テムを設計したり製作したり，あるいは保守する必要のない人々

2. タイムシェアリング・システムの設計，製作，あるいは保守に関係しているか，あるいは関係しようとする人々

これら2つのグループによって本書を読む上で必要な読み方や背景は異なってくる．最初のグループに要求される背景はプログラミングの経験が少しあり，ディジタル型計算機の構造について一般的な知識をある程度持っていることである．そして本書の読み方は，各節が独立に書れているので，詳細は例題の部分をとばしても良い．

2番目のグループに要求される背景はさらに広い．このグループの人達はここにある概念を理解できなければならず，さらにその概念が実現される上での詳細なプログラミング機構を理解できなければならない．この要求はハードウェア論理やレジスタ操作，ならびに基本的なリスト処理やテーブルの作成，参照，および後入れ先出し (last-in, first-out) リストなどについてある程度精通していることを示している．このグループに対する本書の読み方は一度詳細な例題部分をとばして速読し，その後注意深く再読することである．外国語を習得する場合にたとえれば，最初のグループは外国語の読解力を必要とする人と同じであり，また第2のグループは外国語の読み書きができ，そして話せなければならない人と同じである．

別の見方をすれば，このテキストは次のようなレベルの人に読まれるように書かれている：

1. ソフトウェア・システムの設計に関する学部課程用の主テキストならびにシステム設計概念を紹介する上級プログラミング課程用補助テキストとして．上級プログラミング課程を習得していることが望まれる．練習に学生はオペレーティング・システムで必要な待ち行列の取扱法あるいはテーブル操作法のような機構をプログラムしてみるのがよいだろう．
2. 計算機機構の学部課程用の補助テキストとして．このような課程で利用する時は，学生はプログミング経験が必要であろう．本書はこのレベルでは単科大学や総合大学のみならず技術研究所や企業の教育課程でも使える．
3. システム設計およびプログラミングの大学院課程用の主テキストとして．本書をこのレベルで利用する時は，詳細設計における問題点にさらに精通するために，特定のシステムに関する文献および刊行物を本書の題材として用いるようにする．また学生は本書に載っている概念を拡張

し，応用し，そして批評する能力を養うために，プロジェクトに参加して実際のシステムやシステム・モジュール設計を行うことが望まれる．

4. 計算機機構の大学院課程用の主テキストあるいは補助テキストとして．この場合，本書は特にソフトウェア概念を紹介する上で有用となる．ハードウェア設計者も効率的なシステムを設計しようとすれば，ソフトウェア概念にも精通している必要があるのは明らかである．

5. 経験を固め刊行物を読む能力を増強したいと思う職業プログラマ，設置および EDP 管理者，ならびにハードウェア設計者のための独学書として．このレベルでは，読者にある程度システム・プログラミングやハードウェア設計の経験があれば，容易に読破できる．あまり経験のない職業人でもいくつかの詳細例をとばして読めば，非常に為になる．このレベルでは，仕事あるいは管理をしているシステムで使われている基本的な概念を理解したい職業人，特に管理者に役立つ．

本書は3部と7章からなり，それぞれ次のように構成されている．序論である第I部は1章で構成されている．第1章でタイムシェアリング・システムが開発されたいきさつとシステムの主要な設計目標および問題点について述べ，本書を通して必要な基本的概念を招介している．

第II部はタイムシェアリングに利用される計算機の中で重要と思われるハードウェア概念を紹介している．この部は3章で構成されている．第2章はベース・レジスタ，ページングおよびセグメンテーションのような記憶装置のアドレス方式や割付けを詳しく述べ，その例を載せた．第3章は記憶バスおよび制御機構の設計，入力／出力の取扱い，遠隔端末とのコミュニケーションの取扱いに関係した概念について論じている．第4章ではハードウェアによる記憶および制御保護，割込みシステム，ならびに制御ユニットを設計する方法としてのマイクロプログラミングについて考察する．

第III部でユーザならびにシステム・リソースの構成，割付け，および保護に関するソフトウェア概念を紹介し，またその他の重要な設計上の話題について論じる．この部は3章からなる．第5章ではハードウェア・プロセッサおよび記憶装置のスケジューリングや割付けの問題を扱う．また，ソフトウェアによる保護機構についても論じる．第6章で容易なアクセス，共用，保護，およびこれらを達成するための概念といったファイル・システムの設計目標について論じる．入力／出力動作の取扱いについてさらに詳しく論じ，ユーザに対するインタフェースについても若干載せた．第7章では設計の初期段階から測定，

評価，信頼性，回復性，および保守性を考察することの重要性について論じる．そしてこれらの領域に含まれる概念を載せた．

巻末にタイムシェアリングに関する包括的な文献目録を付した．

システム・プログラミング課程においては，第II部の題材は他の本や演習問題を追補せずに使うことができよう．第III部は学生が利用できるシステムの例ならびにプログラミングおよび設計演習を追加するようにすればよい．大学院課程ではその他の文献を追補するのがよかろう．

計算機機構課程では，おそらく第III部の内容は他の本を追加しないでも使えるだろう．第II部はハードウェアの詳細例を引用したり，データ経路やレジスタ機構，演算アルゴリズム，制御設計など本書に含まれていない重要な課題を補なって読むのがよかろう．

本書の内容はCommunications of the ACM誌の1968年3月号で発表されたACMカリキュラム委員会報告，カリキュラム68，で推奨された2課程に非常に深く関係している．これらの課程はI4，**システム・プログラミング**，とA2，**上級計算機機構**である．本書はまた課程I3の**計算機機構**にも関係しており，そして課程I1の**データ構造**で考慮される重要な題材も含んでいる．本書を読むにあたって有益な基礎を与える課程はB1，計算の基礎，とB2，計算機とプログラミングである．本書はまたArdenのPh.D.プログラミング・カリキュラム* の課程III，**システム設計**にも関連している．

本書はハードウェアおよびソフトウェアに関連した部分で構成されているが，全体を通してそれらの密接な関係が強調されている．実際にこの密接な関係はハードウェアを論じる部分でいくつかの重要なソフトウェア概念を紹介するのに役立っており，また逆も同じである．技術が進歩し，知識が向上するにしたがって，ハードウェアおよびソフトウェアの境界は移動するであろうが，基本的な概念や問題点は残るであろう．

本書に載せられた概念や機構は一般に著者が考え出したものではない．これらタイムシェアリングの開発されたものが誰の仕事に属しているかはすべて参照をつけた．私の書いたものが特定の著作と非常に似ている場合はその著者名をあげて明白に参照を付した．他の領域で身近なグループの考えを基にした題材については番号によって参照をつけた．各章の終りに記載した参考文献は私の知る限りその章の題材の出典となるものである．他の人が考え出した概念に

---

* B. W. Arden, The Role of Programming in a Ph.D. Computer Science Program, *Commun. ACM*, vol. 12, no. 1, pp. 31-37, January, 1969.

ついて参照をつけると同時に著者の考え違いや明確さを欠いた部分に対して全責任を負う.

本書の出版にあたり援助していただいた多くの人々に感謝の意を表したい. 上に述べたように, タイムシェアリングおよびオペレーティング・システムの分野で仕事をしている人々, 特にかつてCalifornia大学におり, 現在Berkeley Computer Corporationの研究者であるL. P. Deutsch, B. W. Lampson, W. W. Lichtenberger, M. W. Pirtle, とその同僚, ならびにMITのF. J. Corbato, R. C. Daley, P. J. Denning, J. B. Dennis, J. H. Saltzerとその同僚の考え方には終始恩を受けた. また本稿を読んで有益な助言をしていただいたM. D. Kudlick, E. A. Feigenbaum, J. H. Saltzer, M. W. Pirtle, B. W. Lampson, そしてK. Siberzに感謝する. Shell Developmentでの共同研究者, 特にM. D. Kudlick博士とは価値ある討論を行なった. このような仕事の必要性を最初に認めたShell Development Companyの管理者, 特にRobert S. Miller博士, とF. G. Stockton氏には深謝する. 本稿のいくつかの題材は最初にShell Developmentの技術報告に載せられたものである. この題材の出版を許可したShell Developmentに感謝する. 私の原稿を注意深くタイプして下さったSharon Overtonに特に感謝する. 最後に本書の執筆中に心配をかけた妻と娘に感謝したい.

RICHARD W. WATSON

訳　者　序　文

タイムシェアリング・システムがもつ意義が指摘され，その有用性が実証されたのは今から10年ほど前のことであるが，それ以来数々のシステムが研究開発されている．タイムシェアリングの機能は最近の計算機にとって最も重要な基本技術であり，応用分野にとってももはや常識的なものとなっている．

訳者も電気試験所時代にタイムシェアリング・システムの開発に携わったことがあるが，そのささやかな経験からいえることは，タイムシェアリング・システムは最近のハードウェアおよびソフトウェア技術の集大成であり，これからの使用経験をもとに更に改善を重ねる必要が残されていることである．つまり将来の計算機技術の一端はタイムシェアリング・システムのユーザの立場からハードウェア的にもまたソフトウェア的にも検討がなされ，体系化が進むものと考えられる．

幸い，著者はタイムシェアリング・システムとして最も評価の高いカリフォルニア大学バークレイの XDS 940 とマサチューセッツ工科大学の Multics システムの造詣が深く，そこに含まれる重要な機能ならびに機構をユーザの立場で解説しているので，読者は容易にその問題点を理解することができるであろう．タイムシェアリング技術で最も重要，かつ本質的な課題をこれだけ要領よく解説し，まとめている本は他にないであろう．米国の大学において，本書が教科書として広く活用されている理由がうなづける．

読者の皆さんには，タイムシェアリング・システムという観点からも，またハードウェア・ソフトウェアの機能分担の改善という将来の計算機に課せられた大きな問題点からも本書はお役に立つものと期待できる．

本書の翻訳ならびに校正に際しては慶応義塾大学大学院の由良英一君に多大の御協力をいただいた．本書はむしろ同君との共訳であったというべきかも知れない．また，同じく慶応義塾大学大学院の吉川良一君，清水洋壱君，山下哲男君，ならびに石塚朝生君をはじめ研究室の同僚に何かとお世話になったことを付記し，御礼申し上げたい．

最後に本書の出版に際し，長い間，訳者のわがままを甘受し，御協力を惜ま

なかった日本コンピュータ協会の村越勝弘氏の御好意に対し深く感謝の意を表したい.

昭和49年秋

相 磯 秀 夫

目　　次

第 I 部

概　　要

第 1 章

汎用タイムシェアリング・システムの概要

1.1 タイムシェアリング・システムの動機

1.1.1 概　　要

システムを設計しようとする人は，そのシステムが実行する機能と動作する環境に対する影響について，明確に理解している必要がある．この理由から，汎用タイムシェアリング・システムの開発をもたらした動機やその環境に与える影響の特性のいくつかを調らべてみるのは有益なことである．

プログラム内蔵型ディジタル計算機システムは，巨大な計算機から汎用情報処理システムへと成長してきた．[28†] これらのシステムの能力は，数の単純な記憶，検索，そして変換から，最も一般的な形の記号システムや構造の記憶，検索，変換へと発達してきた．いくつかのこの発達の関係について，次節で述べる．

1.1.2 初期のバッチ・システム

初期のシステムでは1人のユーザがプログラムのデバッグや実行の間中，コンソールにすわることを許していた．システムがさらに複雑になり，計算業務の数が増大するにしたがって，ユーザをコンソールの前にすわらせるといった機械とユーザ間の速度の不整合は，不経済であることが明らかになってきた．これらのシステムは，ユーザがプログラムを用意して，カードのような入力媒体にプログラムを穿孔し，それをオペレータに渡して実行を依頼するようにすれば，さらに効率的に利用できる．オペレータは多くのユーザのカード・デッ

† 番号をつけた参考文献は各章の終りに載せてある．また巻末にはわかりよい参考書の一覧を置いた．

クをひとまとめにしてそれをカード・リーダに入れることになる．それから，**オペレーティング・システム**（operating system）と呼ばれる，オペレータを手助けするように書かれた計算機内にあるプログラムが一度に1つのプログラムを機械の中に読込み，それを実行し，結果をプリンタに書き出す[29]．オペレータは印字された出力を検索して，それを入力カード・デックと一緒にユーザに返す．ユーザが機械に接したのは，カードを渡し，出力をもらった計算機センターの窓口を通してのみであった．

このような**バッチ・オペレーティング・システム**（batch operating system）は，さらに多くのユーザをサービスし，高価なシステム・リソースをさらに効率的に利用するように絶えず改良されてきた．しかしながら，このようなシステムの根本的な設計方針は，一般にユーザ・リソースの効率的な利用というよりはむしろ，計算機リソースの効率的な利用という方向である．以上のような利用方法では計算機センターにカード・デックを託けて，出力結果を返してもらうまでに時間の遅れがある．この時間の遅れ，すなわち**ターン・アラウンド時間**（turnaround time）は，初期のシステムではどこでも，計算センターからユーザまでの距離，センターの混み具合，計算機センター・システムの特性によって，2時間から数日にもなった．

バッチ形式の処理は，多くの型の**業務処理**（production run），すなわち，何度も使われるデバッグ済みのプログラムに対するデータ処理に適している．多量の計算をする仕事もこの範囲にはいる．プログラムを書き，デバッグしている間では，このような処理形式は，しばしば，不満足であるといわれていた[1]．プログラムを準備する過程でのセミコロンの入れ誤りは，数時間から数日間の時間を無駄にする．プログラムが実行できるまでに，この種の遅れが数回ある．一度プログラムが実行されると論理的な誤りを取り除くために，さらに多くの労力とプログラムの実行が必要である．このプログラムを書き，デバッグする間のいらだちがタイムシェアリングに対する最初の動機を導いた．

1.1.3　タイムシェアリングの概念

タイムシェアリングの分野における初期の研究者たちは，人間の思考過程と応答が計算機の論理，演算能力に比較して遅いことから，各々のユーザが計算機にオンラインで端末から会話できるようにして，計算機リソースを1人のユーザから他のユーザへ切り換えることにより，あたかも1人のユーザが独占的に計算機を使用しているかのようにすることができると考えた．その考えは，1人のユーザから他のユーザへ計算機リソースを時分割(タイムシェアリング)

で切り換え，あるいは多重化† することであった．それに対して，バッチ形式の処理では，計算機の実行と実行の間で，ユーザはある**知的活動**（intellectual activity）から他の知的活動に強制的に移り変えさせられる．初期のシステムはタイムシェアリング形式の処理ができることを示した．強力なオンライン編集とデバッキング機能が作られ，それによってプログラマはプログラミングの誤りを発見すると直ちに修正できるようになった[1,2]．

1.1.4 問題解決の手助けとしてのタイムシェアリング

しかしながら，タイムシェアリングの主な利点はプログラマのいらいらを解消することではなく，計算機の手助けによる新しい形の問題解決（problem solving）にある[3]．ユーザはコンソールの前にすわり，会話的に何回もプログラムを書いたり，デバッグしたり，実行することができる．このようにして，常に問題と十分接することができ，問題を解くために想像力をより働かすことができる．そして，無駄な労力に要した時間の損失を著しく減らすことにより，より多くのこれらの試みを行なうことができる．さらに，ユーザの要求によりシステムのファイル記憶を使って，情報の更新，検索，そして他のユーザと情報を共用できるようになる．このような機能は計算機の手助けによる問題の解決方法をいろいろと変えている．タイムシェアリングによる計算機の操作は単なる従来のバッチ向き計算機による問題解決の手助けや意志決定の拡張だけではなく，十分その利点をいかすことにより，質的向上の出発点になっている．この質的な改革は人間の思考過程が**知的**（intellectul）と**直観的**（intuitive）といった2つの基本的に異なった形で現われることによるものと思われる．知的思考過程は大部分意識的であり，規則正しく順序に従って進むように思われる．このような過程は，完全な数学的証明といったような伝統的な論理システムによってモデル化される．一方，直観による思考過程は主に無意識であり，並行的に進み，規則的でないが，洞察力により飛躍的に進む．完全な数学的証明は，意識的であり，順序に従い，規則性があるが，数学者が洞察力でその論理が真であるとする思考過程とその証明の概観は規則的でなく，またすべて意識的ではない．このような洞察力は突然現われる性質がある．洞察力と直観は養うことができるが，それらの出現を保証することができる方法は存

† **多重**（multiplex）という用語は，通信工業から借りてきている．そこでは，1つの通信回線で，多くの通信を送信することに使われている．ここで使われている多重という用語は，いくつかのプロセスが同時に使う効果を生み出すように計算機システム・リソースを共用することを示している．

在しない．直観と知識はともに必要である．直観力は思考過程を導き，示唆し，そして知識はそれを確かめ，吟味し，ギャップをうめ，結果を他方へ伝える．

われわれは計算機の機能が問題解決や意志決定における知的思考過程の手助けに有効であることに気が付いた．人の直観は問題の解決と決定を促し，問題を解く人は検査過程を手助けするようにプログラムを書く．バッチ処理では，ユーザは各プログラムを作り，実行するための時間や労力に多くの費用を費やす．問題を解こうとする人はその努力で有効な結果を導くことを確かめたくなるので，バッチ処理ではいかにして計算機を使うか，あるいは計算機を使うべきかを決めることよりも，問題の解析と調査に全勢力を費やすことになる．バッチ処理での問題解決は，知的，解析的で順序的な思考を行う傾向になる．

一方，タイムシェアリング・システムでは，ユーザは自分の行なう情報処理に対して時間的な制約がなくなるので，グラフィック・コンソールを使って計算機と会話をしながら調子を合わせ，仮定や試みをいちいち吟味しなくてもよくなり，多くの無駄を減らすことができる．その結果として，タイムシェアリングを利用すれば，その利点を十分生かすことによって，より自由に直観的思考が行なえる．このようにタイムシェアリングによる計算機の利用は，バッチ処理によるものとは質的に異なった使い方になる．

タイムシェアリング・システムから得られたいくつかの利点は，新しいバッチ処理に生かされている．本書に記載されているものと同等あるいは同一の多くのシステム設計技術を使って，より新しいバッチ・システムは，ターンアラウンド時間を数時間から数十分に短縮できる．このような新しいバッチ向システムはまた，情報ファイルの共用に重みを置いている．その結果として，このようなバッチ・システムのユーザは他のユーザとより自由に実験し，より容易に共同作業を行なうことができる．

1.1.5　タイムシェアリング・システムとバッチ・システムの比較

タイムシェアリングと問題解決についての考え方を支える客観的な証拠は何であろうか．タイムシェアリングとバッチ処理を比較した研究は多く行なわれてきたが[4,5]，この種の研究は難しい．その研究におけるバッチ処理は，ターンアラウンド時間が数時間といった古いものであった．主観的には，ターンアラウンド時間が30分以下といったもっと速いバッチ処理では，タイムシェアリングにずっと近い問題解決形式が可能であるという感じがする．

大部分の研究はむしろ狭い視野で行なわれたものであり，終了するのに40時

間から60時間かかるような比較的難解なプログラミングの仕事を実行する際の価格との比較に対してのみ行なわれている．タイムシェアリングを利用するとその仕事に対する労力を，もしプログラマがタイムシェアリングを十分経験しているとすれば，約20%のマンパワーを節約することができる．タイムシェアリング・システムを使う際の経験は，最大の効率を得るために非常に重要であるということを見逃すべきではない．この発見は，1人1人が経験することである．与えられた道具を十分いかすための新しい形式の問題解決法を開発するためには時間と経験が必要である．一般にタイムシェアリングのユーザはバッチ処理の下で同じプログラムを書き，デバッグするのに必要な時間以上の計算機時間を使っていた．ユーザ間でプログラマの個人的な能力差が大きいため（10対1に及ぶ）きちんと実験するのは難しい．

われわれが知る限り最も興味深い研究は Gold によって行なわれたもので[5]，彼は価格比だけでなく，問題解析に関するその他の要因についても興味をもっていた．たとえば，問題解決の効率や，タイムシェアリングとバッチ処理で得られる問題の理解のレベルに関するユーザの感覚といったようなものである．Gold の実験に基づく問題は，インダストリアル・ダイナミックスを専攻する学生に産業とその市場構造についてのシミュレーション・モデルを開発させることである．学生の仕事は，そのモデルの操作，パラメータおよび制約をまず学び，シミュレーションしているマーケット内の小さな独立企業家の利益を最大にするように，現在産業界で使われているそれらの要因を決定する新しい方法を導くことである．Gold は問題を解くために必要なマンパワーと計算時間に関して行なわれた他の研究と同じような結果を得た．すなわち，タイムシェアリング・システムを使用して平均16人－時間かかる問題を解くのに，バッチ・システムでは平均19.3人－時間の労力が必要であるが，CPU 時間はタイムシェアリングで平均5.74分であるのに対し，バッチで平均1.25分である．現実の状況でその仕事をする人々への給料と計算機使用料を考えると，総計の費用はタイムシェアリングの方が安くなるだろう．われわれの観点からみて最も重要な発見は，タイムシェアリングのユーザはバッチのユーザよりも高い利益水準を達成し，問題に対してより深い認識と理解をもつことである．結論を固めて異なったシステムに対する相対的な利点をあげる前に，さまざまな計算機システムを使った人間の問題解決に対する行動についての研究がさらに必要であるが，Gold の研究や多くのレベルのタイムシェアリングにおけるユーザの受け入れをみると，上であげたタイムシェアリングの主観的動機づけに異論がないことがわかる．

1.1.6　プログラムおよびデータの共用

タイムシェアリングあるいはバッチ・システムにかかわらず，データおよびプログラムの共用の概念についてここで述べておくべきであろう[6]．計算機システムをユーザの社会における技術や情報の貯蔵庫として働かせ，それらを遠隔地から出し入れできることは，すばらしい構想である．ユーザはカード・デックの複製を得ることによって常にプログラムやデータを共用することができるが，多くの遠隔地点から直接的に情報を取り出せる場合とは質的に異なる．このような機能を備えたシステムは新しい共同作業の方法を与え，個人の労力に対しての効果を増大させる．そのような機能をシステムに設けることは，設計や設備を複雑にする．アクセス法やバックアップに対する考慮が払われなければならない．プログラマやデータを共用する際には，あるレベルの機密を設ける必要がある．この機密を守るために，プログラムやデータが広く共用される前に，そのプログラマ以外の人により検閲あるいは特別の検査が必要になるであろう．機密を保つために，技術を公開し記載する技術出版物は，通常出版される前に検閲される．また，計算機のプログラムやデータが広く共用される前にさまざまな段階の検査が必要であろう．**プログラム・ライブラリ**という用語はある時期に共通に使われる共用プログラムの集まりを意味するが，一般的に言えば，プログラム・ライブラリやライブラリの印刷物には注意が払われていない．

1.1.7　ま　と　め

タイムシェアリングの動機は計算機とユーザをより密接にすることであると指摘してきた．それはユーザがオンライン端末から会話的に計算機を操作でき，データ処理への要求に対する素早い応答ができるからである．

強調したいことは，速度と機能が拡大するにしたがって，設計者の注意深い配慮に値する重要な質的変化が起こるという考え方である．このような質的な変化に気づくと，設計者は最初に解き始めたものとは全く異なった問題を解いていることを見抜ける．その結果として，常に設計者は設計目標を新しい状況に合うようにしなければならない．さらに速い応答とより広い情報の共用がタイムシェアリングと新しいバッチ・システムで可能になり，これにより計算機を使った新しい形式の利用方法と重要な新しい応用分野が開けてくる．

1.2 タイムシェアリング・システムの形式

1.2.1 概　　要

システムにあるすべてのリソースをさらに効率的に利用するためにオペレーティング・システムが開発されてきた．**リソースの共用**（resource sharing）は基本的な概念であり，さまざまな階層のオペレーティング・システムは，次に挙げるシステムのリソースのどれに重みを置いているかによって分類することができる：

演算論理装置
主記憶
補助記憶（ドラム，ディスク，その他）
周辺機器（カード・リーダ，プリンタ，その他）
通信機器（通信回線，遠隔端末）
コンパイラ
データ・ファイル
サービスおよび応用ルーチン
プログラミングおよびデバッグ時間
ユーザ使用時間

オペレーティング・システムには，たった1つのプロセスだけが使うことのできる種々のリソースを共用し，多重化するものがあり，そのようなオペレーティング・システムの総称を**タイムシェアリング・システム**と呼ぶことができる．リソースの共用が時間的に逐次であるので，このような広範囲のシステムがタイムシェアリングと呼ばれる所以である．本書では，タイムシェアリングという用語を次のシステムに限定する；そのシステムではユーザがシステムの第1のリソースであり，ユーザが端末からオンライン（直接接続された端末）で交信できるようにし，その結果，高速の応答が得られるようにして効果的にユーザをシステムの一部とみなすようにする．この定義によれば，広範囲のシステムがタイムシェアリングという用語にあてはまるので，さらに細かい定義と分類が必要になる．

1.2.2 オンライン・ファイル保守および検索システム

これらのシステムは非常に特殊目的用であり，その特長は共通の情報ベースへの情報の追加や問合わせの範囲に制限があることである．大部分のこの種のシステムでは，ユーザがプログラムを作ったり，決められた以外の問合わせを許していない．この種のシステムの例としては，航空会社の座席予約システム，バンキング・システム，証券情報システム，オンライン在庫管理システムおよび制御システムがある．有名な実例として，American Airlines の SABRE システムがある[7]．

1.2.3 専用タイムシェアリング・システム

これらのシステムの特長は，極めて限られた数の言語（通常1つ）だけをユーザに使わせ，その実行を許すことである．ファイル操作の要求はほとんどなく，ファイルの構造も単純である．この種のシステムの例としては，Rand Corporation の JOSS システムや IBM の Quiktran システムがある[8]．

1.2.4 汎用タイムシェアリング・システム

これらのシステムにはいくつかの特徴がある：

1. 通常20あるいはそれ以上の多数のユーザに対して速い応答時間をもつ．
2. テキスト・エディタやデバッキング手段，アセンブラ，種々のコンパイラ，特殊応用ルーチンのライブラリといった巾広いオンラインのソフトウェアが同時に使える．
3. ユーザがプログラムやデータを格納したり，簡単にアクセスしたり，共用できる大容量記憶装置を用いたファイル・システムを備えている．
4. 付加装置を加えるための拡張機能がある．

これらのシステムの例として，MIT の Multics(multiplexed information and computing service) システム[3] や System Development Corporation の Q-32[2]，そして，California 大学 Berkeley の XDS-940 システム[9] がある．

1.2.5 オンライン・アクセス可能なマルチプログラミング・バッチ・システム

この種のシステムは先に述べた汎用タイムシェアリング・システムの特性を備えているが，リソースを共用するアルゴリズムの設計目標がオフラインで作られたプログラムの処理が最短時間で終結するように考えられている点が異なっている[10]．このような設計目標の結果として，与えられたハードウェア構成では，比較的少ないオンライン・ユーザに対して適当に応答するようになっている．多数のユーザに対しては別の設計目標がとられるはずである．プログラムの作成，実行，およびファイルの扱いに関してもまた制約がある．この種のシステムの例として，計算機メーカの提供するオペレーティング・システムである IBM の OS/360，UNIVAC の Exec 8 ならびに GE の GECOS がある．

新しいバッチ専用システムと汎用タイムシェアリング・システムの主な共通点は，**マルチプログラミング**（multiprogramming）の採用である．すなわち，1つの計算機システム内で2つ以上のプロセスを同時に実行できることである．同時とは2つ以上のプロセスが部分的な状態で完結しており，リソースがこれらのプロセス間でシステムの負荷の均衡をとり，応答時間やスループットの要求を満足するように割当てられることを意味する．マルチプログラムでないシステムでは，次のジョブが始められる前に1つのジョブが終了する．マルチプログラミングという用語を**マルチプロセシング**（multiprocessing）と混同してはいけない．マルチプロセシングという用語は2台以上の中央演算処理装置（演算論理装置）を持つように設計されたシステムに使われる．

先に述べた種類のシステムは，設計・製作の基本となる概念の多くが共通に使われている．1種類のシステムの設計概念の詳細を理解することは，他のシステムを勉強する際の確実な基礎となる．本書は汎用タイムシェアリング・システムの基本的な設計の要求事項と概念について論じる．本章の残りで，この本を通じて使われる概念について明らかにし，いくつかの設計規則とタイムシェアリングの分野での経験から開発されたシステムの要求事項の概要を述べることにする．本章あるいは後の章にある設計方針あるいはシステムの要求事項は，将来のシステムに絶対的にあるべきものとは言わないが，1人の経験による技術的な判断と解釈によるものであることを心に留めてほしい．独自の設計方針をたてるため，あるいは，現在の設計における問題を勉強し始めるための基礎になり得るものとして，非常に参考になるであろう．

1.3　設計上の要求事項と問題点

1.3.1　本質的問題点と技術的問題点

タイムシェアリング・システム，あるいはどのようなシステムにおいても，設計上の要求事項と問題点について考えてみると，Saltzer[11] が提唱しているように2つのクラスにそれらを分けることが有益である．すなわち**本質的**（intrinsic）と**技術的**（technological）なクラスである．本質的な要求と問題点は問題自体に固有のものである．技術的な要求と問題点はその時点での技術に固有であり，ある経済的な状況に本質的な要求を適合させるためにその技術を使おうとするときにもち上がる．技術が進歩するにしたがって技術的な問題点は消滅したり変化したりする．

汎用タイムシェアリング・システムの設計に関する本質的な問題点は，人と人とのコミュニケーション，アルゴリズムや情報の共用，そして人と計算機とのコミュニケーションのあたりに集中している．タイムシェアリング・システムのユーザは多勢いる．各人が自分自身のデータとプログラムの格納領域を所有している．2つの重要な本質的な問題点は，(1)決められた手法で情報のアクセス，転送，共用が容易にでき，(2)この情報がシステムあるいはユーザの誤りに対して保護されていることである．

汎用タイムシェアリング・システムにおける技術的な問題点は，先に述べた機能を数台の処理装置や主記憶，補助記憶および種々の入出力（I/O）機器からなる与えられた計算機システムに組み入れることに集中している．多くのユーザに1台の計算機システムを共用させることに興味がある理由は，この方法が現在および将来で経済的に本質的な問題を解決するただ1つの方法であるからである．ユーザにそのような機能とアクセスの容易さを提供するために必要なハードウェアとソフトウェアは莫大なものである．ユーザやその計算がすべての装置を連続して利用することはない．そのため，1つの目標は多くの計算にすべてのリソースをバランス良く使用することである．1つの計算が入出力を行なっている間に，他のものは計算を行なうことができ，そして3番目の計算はユーザがコンソールで考えている間待っている．

本書では，コミュニケーションおよび共用といった本質的な問題点と，プロセスが同時に実行される際のリソースの割付けといった現在あるいは近い将来

での技術的な問題点の双方に関して議論する．ここで概説する概念の多くは現在ソフトウェアで作られているが，将来のシステムではハードウェアあるいはハードウェアの助けを借りて実現されるであろう．しかし，基本的な問題点はそのまま残るであろう．

1.3.2 ファイル・システムと情報の共用

設計者の仕事の1つは，システムに組み入れる本質的な目標を定めることである．中心となる目標の1つは，個人の情報を記憶し，また他人と情報を共用する媒体となるファイル・システムを作成することである．

ファイルはユーザに対して最も目につくシステムの一部分であり，ユーザが使える最も重要なリソースの1つである．**ファイル**（file）は名前で関係づけられた情報の集まりである．ファイルは字，ビット，計算機語といった要素の列として記憶される．ファイル・システムを設計する際に関係する本質的な問題点は，さまざまな種類の情報を転送し，検索し，保護し，共用し，そして記憶することに集中している．言い換えれば，ファイル・システムが，マニラ紙の封筒，ファイル・キャビネット，回転式カード・ファイル，あるいはどのような手段で作られようとも，読者になじみの深い良いファイル・システムのすべての要求事項をそのファイル・システムが満たしていなければならない．

融通性をもたせ，記憶，検索の速度を上げるためには，記憶された情報を素早くとり出すことのできる機器が必要である．さらにユーザに何をどのくらい記憶するかについて，きびしい制約を感じさせないように適度な価格の十分な記憶領域をもつべきである．現在および近い将来，この要求を満たすためには，ディスク・パックやテープのような取りはずし可能な記憶媒体を含む機器の階層化が必要であろう．技術的問題点はあまり使われない情報は低速度で大容量の機器に移されるような機構を作り，すべてのユーザの機能を全体としてハードウェアに無関係にすることである．また，ユーザが自分のファイルがどこにどのように記憶されているかを知らなくても，文字や数字のファイル名で参照できるようにすることが必要である．

機密や安全性に対する本質的で技術的な要求は，融通性と速度とに対する要求と同様に重要である．ユーザが共同作業を進めるためには互いに他のユーザのプログラムやデータ・ファイルを共用させるべきであるが，ファイルを作る特定のユーザがその人のファイルを誰にアクセスさせるか決めたり，各個人あるいはグループの人に対してファイルのアクセス方法（読出し専用，読み書き可能，実行専用）を規定できるようにすべきである．もちろん，後者の要求は

ファイル作成者自身だけがアクセス可能な個人用ファイルを持つことができることを含んでいる．アクセスを許可されたユーザが誤ってファイルを削除したり，損傷を与える可能性があるので，このような場合に備えて，バックアップや回復の手段を備えておくことが望まれる．ここで述べた重要事項は本質的なものである．

共用する情報を多数複写する問題について考えなければならない．1つのファイルの複写を更新するときに，同様に更新するために他のすべての複写ファイルを捜し出すことは容易ではない．プログラム・ファイルから論理的誤りを取り除いたり，ある点で改良を加えたプログラム・ファイルはその良い例である．また，主記憶や補助記憶領域の節約に関連して，システム内で共用する情報の複写数を最小にしたいという技術的な理由も存在する．プログラムやデータの1つを完全な一般性をもたせて共用させると，技術的問題を生み出す可能性もあるので，与えられた技術レベルにあわせて，その共用をいかに取扱わせるかに関する妥協が必要である．

ユーザのすべてのファイルは，信頼性があまりないと思われる一群の装置に格納されるので，ユーザが誤って他人のファイルをアクセスしないように，また，システム・ソフトウェアやハードウァアがファイルに損害を与えた場合にバックアップを保証するような十分な手段が講じられていなければならない．これらの問題点は技術的なものである．ファイル・システムとその他のI/Oについては第6章で論じる．

1.3.3　システムとのコミュニケーション

2番目の本質的な目標は，システムとのコミュニケーションに関するもので，設計者は端末設計，計算機から端末への情報転送，端末間のコミュニケーションおよびシステムとのコミュニケーションのための言語設計などを考慮する必要がある．

後者の要求は，(1)システムとコミュニケーションのためのコマンド言語や通訳プログラム，(2)プロセッサを制御するプログラムを書くためのプログラミング言語や飜訳ルーチン，(3)上の手助けとなるためのプログラム・ライブラリ，についての設計が必要となる．コマンド言語は，ファイル・システムやテキスト・エディタのようなサブシステムへのアクセス，あるいはデバッグ手段や言語飜訳プログラム，および他のシステム・リソースとのコミュニケーションに使われる．コマンド言語は他の言語と同様に注意深く設計されるべきである．（第6章参照）

端末の設計はそれ自体特殊な課題であり，本書では最も一般的な考えのみを論じる．利用できる端末の種類は増大するであろうから，タイプライタ，グラフィック端末，遠隔計算機，遠隔カード・リーダ，プリンタといった巾広い種類の端末機器が接続できるような設計にしておくべきである．各端末の情報転送速度は異なる．遠隔地点への情報転送もまた1つの大きな課題であり，ディジタルおよびソフトウェア・システム設計者にとって有用と思われる論理的概念のみ第3章で紹介する．端末のコミュニケーションに関するソフトウェアの問題については第6章で論じる．

1.3.4 信頼性と回復性

第1の本質的な目標は信頼性の高い動作である．ユーザは，スケジュールされた有効時間の間，システムが正しく動作していることを期待する．もし故障が起こったときは，できるだけ円滑にそして速やかに回復してほしいと願う．そのためには故障が起こった時点でのシステムの状態にできるだけ近ずけるようにシステムを回復させる必要がある．システム設計者は，ハードウェアおよびソフトウェアの故障が設計あるいは製作段階で細心の注意を払ったにもかかわらず起こりうることを仮定しておかなければならない．良い設計では，設計の初期段階からハードウェアおよびソフトウェアの双方に故障検出，故障位置発見，回復手順を組み入れている．この重要な課題については，第7章で論じる．

1.3.5 リソースの割付け

本質的な目標がはっきり定まってから，設計者は現存する技術を使ってどのようにそれらを実現するかを決めることができる．中核の技術的問題は，できればユーザに分からないようにして，1つのハードウェア複合体を多数のユーザに共用させることである．多数のユーザ間で1つハードウェア複合体を共用させる問題について有益な見方を Saltzer[11] が述べている．以下の論議は彼によるものであり，図は彼が描いたものである．リソースを共用するために，タイムシェアリング・システム設計者は1つのハードウェア・システムをとり，ソフトウェア機構を使って任意の数の独立した**仮想プロセッサ**（virtual processor），すなわち擬似か仮のプロセッサを作り出し，それをユーザに割当てることによってその問題を解決している．各仮想プロセッサは次のものを所有している：

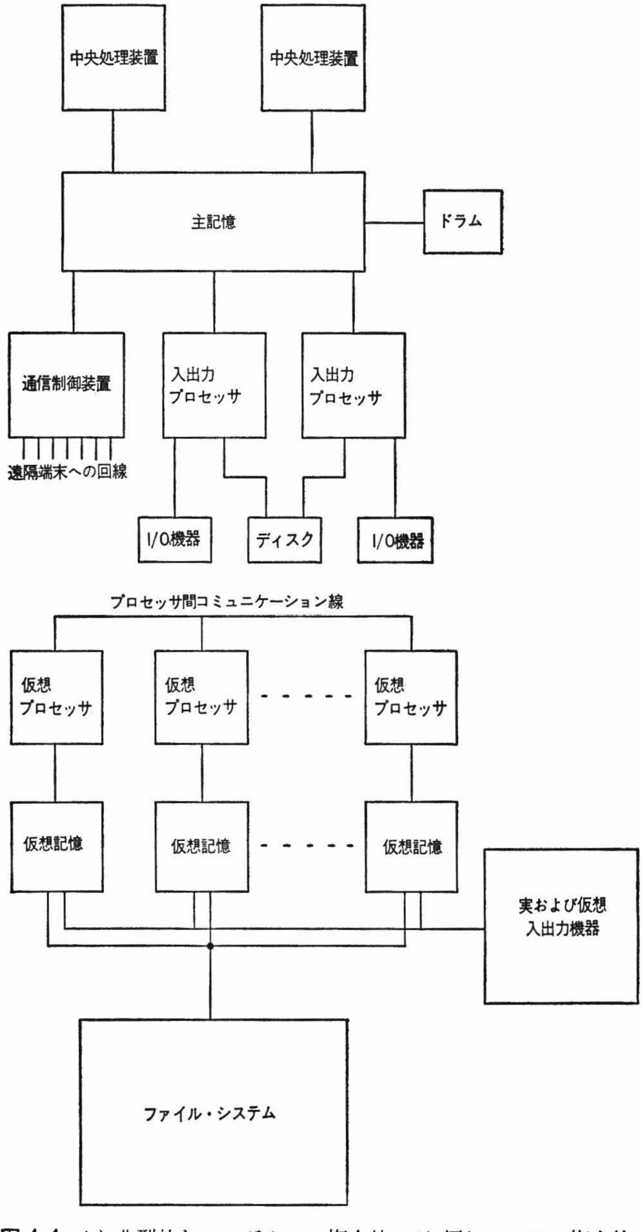


図 1.1　(a) 典型的なハードウェア複合体；(b) 仮想システム複合体

1. ハードウェア・プロセッサの命令あるいはその部分集合を含む命令体系とソフトウェアで作られた付加的機能
2. それ自体のアドレス空間，それは他の仮想プロセッサと共用され得る
3. システム中の他の仮想プロセッサとコミュニケーションするための手段
4. ファイル・システムと呼ばれる大容量情報記憶システムのアクセス
5. 遠隔端末を含む種々のI/O機器とコミュニケーションする能力
6. それ自体を終結する能力

これらの仮想プロセッサは，ユーザが機器の数や種類，動作速度によるシステム構成に注意を払う必要がないという意味で，ハードウェアに無関係になっている．仮想プロセッサは，互いに結合できる独立な計算機システムのすべての特性を所有している．しかしながら，これらのプロセッサはコンソールから簡単に使えるように高度に構成されているという利点がある．最も一般的な形では，上で述べた概念は，各ユーザに自分の問題に適した特性を持つプロセッサで仕事をしているかのような考えをもたらす．言い換えれば，ユーザは自分自身のハードウェア集合およびオペレーティング・システムを持っているかのように思う．360/67 で作られた IBM CP/CMS システムは仮想プロセッサの概念の1つの方法を採用している[30]．各々のユーザは記憶容量やI/O機器を補って，自分自身のシステム 360 構成を規定することができる．制御プログラムは各々のユーザのためにテーブルを作るという意味で仮想的な構成を創造し，ディスクを使って異なった機器のためにI/Oを模擬する．すべての仮想計算機の命令体系はシステム 360 のものである．何人かのユーザは与えられたオペレーティング・システムを共用するかもしれないが，各ユーザはシステム 360 単体上で実行できる自分自身のオペレーティング・システムを所有したり，あるいは自分自身のオペーティング・システムを開発することができる．仮想プロセッサの概念を 図 1.1 に示す．

仮想プロセッセッサを生成してから，それらをユーザとシステムに割付け，システム利用の総計を勘定する必要がある．すべての仮想プロセッサが数の限られた物理的なハードウェア・モジュールを同時に制御することができないので，物理的なハードウェア・モジュールを仮想プロセッサにスケジュール，あるいは割付ける技術が開発されなければならない．いくつかの割付けルーチンに共通した設計目標は，各ユーザが処理要求に対して速い応答を受け取ることができ，割付けルーチン自体がシステム・リソースを浪費することがなく，そ

して，一時的に休止している仮想プロセッサがスケジュール時間に大きな影響を与えず，主記憶のような貴重なリソースを必要以上に消費しないことを保証することである．仮想プロセッサは I/O 動作が終結するのを待つために休止するときはいつでも I/O バッファのためにいくらかの記憶領域を必要とする．その他の重要な問題は，仮想プロセッサを他の仮想プロセッサの故障から保護し，そして，システムを仮想プロセッサの故障から保護することである．仮想プロセッサの生成と割付けについては，さらに1.5節と第5章で論じる．

1.3.6　システムのオーバーヘッド，利用率，応答時間

スケジューリングや保護のようなシステム機能を実行するのに消費される時間は，一般に**オーバーヘッド** (overhead) と呼ばれる．この用語は非常にあいまいであり，一般的にオーバーヘッドは良くないといった意味に解釈される．オーバーヘッドの考えをさらに明確にさせるために，先に述べた本質的問題点と技術的問題点の違いを持ち出すことができる．本質的問題点を解決するためのシステム機能は，ユーザの総合的な効率を増大させる．技術的問題点を解決するためのシステム機能は，1人のユーザに高価なシステム価格を他のユーザと共に分担させる．このようにして，ユーザは個人で得られる以上の強力な設備へアクセスすることができる．どちらの観点からみても，システム機能はユーザに価値あるものである．

システム機能に使われる時間を測定しても，それは絶対的な意味はもっていない．提供されるサービスやその時の技術的状況を確めないで20%のオーバーヘッドの良し悪しを言うことは意味がない．しかしながら，設計者が本質的および技術的問題に対する解答の価格を評価したり，潜在的な問題の位置付けを評価するためには，オーバーヘッドは有用な目安となる．確かにシステムが実行する機能のレベルに関連してシステム機能に費やされる時間を最小にすることが望まれるが，ユーザの仕事全体からみたシステム機能の有用性も理解しなければならない．

情報を共用することが多いシステムにおいては，システムが保護機能に費やす時間は増加してくるに違いないが，情報を共用することによりユーザの生産性をすばらしく増大させることが期待できる．同様に，大容量の仮想記憶領域を使うことはシステム・ルーチンで費やす時間を増大させるであろうが，大規模な応用システム設計にとっては好都合である．許容できるオーバーヘッドはユーザを含む計算機の動作に関連した全価格を十分評価した上で決定できる．仕事の性質に応じて許容できるオーバーヘッドは，計算機設備によって異なっ

てくる.

システム利用率の目安の1つであるシステムの**休止時間** (idle time) はオーバーヘッドより意味がある. 休止時間の総計は, ハードウェア要素の動作特性相互間, およびそれとリソース割付けアルゴリズム間の全設計のバランスを直接示している. Nielsen[12,13] による初期の IBM 360/67 システムにおけるシミュレーションは, 明らかにバランスの悪いシステムから出た休止時間を示している.

オーバーヘッドや装置の利用率の他に, タイムシェアリング・システムに関連してしばしば議論されるパラメータに**応答時間** (response time) がある. これは例えば, ユーザが端末から**改行復帰** (carriage-return) **キー**をたたいてサービスを要求してから, システムがその要求を実行し終わったことを返事するまでの時間である. タイムシェアリング・システムにおける一般的な動作形式では, ユーザは数秒考え, サービスの要求を出し, 応答を待ち, 応答を受け取り, そしてそのサイクルを繰り返す. 考え, 応答を待ち, 応答を受け取るといった基本的なサイクルを**トランザクション** (transaction) と呼ぶ. ある期間内に人と計算機の双方が実行した実際の仕事の統計は重要である. この統計は多くの主観的な要素を含み, 正確に決定することは不可能である. 応答時間が重要である理由は, ある有効な仕事の単位は多くのトランザクションから成り立っているとみなされるからである. 応答時間は, このようにコンソールを使っている間の有効なすべての仕事に関した1つの目安であるとみることができる.

応答時間は, システムのユーザ数, ユーザ要求の性質, 特殊な要求, およびそのシステム設計の関数である. 一般にユーザは単純な処理に必要なトランザクションに対しては非常に速い応答を期待し, もう少し時間がかかるとわかっているトランザクションに対しては, 少々応答が遅くなっても我慢する.

オーバーヘッドと同じように, 応答時間は測定する状況についての情報を伴わないならば, その数値は意味のない数値である. 平均応答時間はさらに意味深長であり, 一定時間内に集めた統計を基礎にしている. 平均応答時間をさらに有効とするためには, ユーザ数や測定時間での要求事項のような統計的情報を付け加える必要がある.

システム構成要素の利用率 (ハードウェア－ソフトウェア) と時間パラメータの測定は, 最近さかんに行なわれている[14-22]. 情報の流れを阻害することを明らかに示すデータがなければ, さらに経済的で応答時間の速いタイムシェアリング・システムを開発することはできない. 設計の初期段階から測定や評価

について考えておかないと，設計の終了時点では設計のいろいろな面で評価することが難しく，不可能になるであろう．設計者は，設計を最良のものとするためにすべての潜在的な情報を利用したいと考える．さらに設計者は，進行中の設計に適用できるすべての経験を将来のシステムに盛り込みたいと考える．この重要な問題については第7章で論じる．

1.3.7　ハードウェアとソフトウェアの相互作用

システム設計にとってもう1つの重要な仕事は，仮想プロセッサが使うために準備すべき基本的なハードウェアは何か，また，独立の仮想プロセッサ，ファイル・システム，その他のI/Oの生成，割付け，保護を行なうソフトウェアを助けるために必要なハードウェア機能は何であるかを決めることである．ハードウェアとソフトウェアの密接な相互作用は極めて重要である．過去の設計手順はハードウェアの設計を先に行ない，その後でソフトウェアの設計を行なっていた．ハードウェアの設計はソフトウェア等を完全に理解していない人によって行なわれてきたし，その逆もまた言えた．このような方法はもはや合理的ではないように思われる．汎用タイムシェアリングやその他の近代的な計算機システムは確かにハードウェアおよびソフトウェアの双方によく通じている人々によって，より効果的に設計することができる．

われわれの経験では，ハードウェア設計が固まる前に，できるだけ長期間にわたってハードウェアとソフトウェアの設計が並列に進められる方が良いと言える．このようにするには，ハードウェアとソフトウエアの設計に責任がある人々にそれら双方の機能仕様の草案を決めさせることである．ソフトウェア設計者は，どのような付加機能が有益であるか，あるいは，どんな機能が障害となるかといった情報をフィードバックできる．その結果，ハードウェア設計者は現在の技術で望ましいハードウェア機能の内どれが経済的あるいは技術的に実現不可能か，あるいは，簡単に付け加えられる機能は何かを決定できる．このような研究には，シミュレーション，その他の測定および評価手段が役に立つ．タイムシェアリング・システムで有用になると思われるハードウェア機能については，第2章から第4章にわたって論じる．

1.4 体系的な設計方針

1.4.1 概　　要

タイムシェアリング・システムは複雑なもので，その設計，実現の双方に多くの人々の共同作業が必要となる．過去において，複雑なハードウェアーソフトウェア・システムの設計，実現は性能の達成および予測した完成期間の面からは，満足できるものはほとんどなかった．この過去の経験から，将来のシステムを設計し実現する際の方針や提案がいくつか出されている[23-25]．特に重要なものとして，2つの原則があるように思われる：

1. 設計は，いくつかの段階で機能的にモジュール化すべきである．
2. いくつかの基本的な機構を使って大部分の仕事を実行するように設計すべきである．

1.4.2 モジュール性

この2つの原則に従う意味では，ハードウェア設計者の方がソフトウェア設計者よりもずっと恵まれている．例えば，中央処理装置（CPU）を考えてみよう．CPU の主な機能は，記憶装置から情報を取り出して，その情報に対して演算を行なったり，あるいは，他の論理的変換を行なって，その結果を記憶装置に戻すことである．CPU は理論的に，組合わせ回路（記憶素子を含まない論理回路）や順序回路（記憶素子を含む論理回路）理論を使って，ある最小数の単一の記憶素子や論理ゲートから組立てることができる．このような設計はおそらく個々に相互接続された素子群の入り乱れた配列を生み出し，それを理解するのは難しくなってしまう．そこで，設計者たちはモジュール化の手法を使う．CPU は決まった数のレジスタ，デコーディング・モジュール，演算論理モジュール，および制御モジュールを使って設計される．これらのモジュールは数本の明確に定義されたデータ転送回路や制御信号経路により相互接続される．このようなモジュール化の手法により，情報や制御の流れを容易につかむことができる．その結果，モジュールはわずかな数の AND ゲート，OR ゲート，記憶素子（フリップ・フロップ）のような基本論理素子からなるサブモジュールを組合わせて構成される．高度のモジュール性の他に，CPUの設計にはいくつかの基本的な制御機構およびデータ転送機構が使われていることに

注意してほしい．

明確に定義されたモジュールの相互接続から形づくられているシステムは一般に，理解，修正，保守，評価がしやすい．経験からは，各々のモジュールは概念的にまとまったある機能を果たすべきであると言える[25,22]．各々のモジュールは明確に定義された少数の入出力信号，すなわちパラメータを持つべきである．その結果，いくつかのサブモジュールから1つのモジュールを作ることができる．モジュール設計の目標は決められた時間内にプログラムでき，テストできるような大きさにモジュールを決めることである．ユーザは複雑な物理的機器群から出来ている仮想システムと交信するので，モジュール性のレベルの概念が必然的に出てくる．たとえば，ファイル・システムの設計を考えてみよう．いくつかの機能的レベルは本質的な目標を実際の技術的問題に変えている[26]．それらは：

1. 論理的なファイル・システムのレベル．そのファイルは記号名称でアドレスされ，論理レコードから成り立つ．
2. 物理的ファイル・システムのレベル．物理的な機器アドレスと物理レコードから成り立つ．
3. 多くのアクセス要求の中から，機器やファイルの特性および状況を最もうまく利用するためのアクセス・スケジューリングを決定するレベル．
4. 個々の機器をアドレスし，転送を行なうレベル．

各レベルはいくつかの機能的タスクからできている．そのタスクは次のより高いレベル，および低いレベルとコミュニケーションしなければならない．このような形で全体の設計を行なうことができれば，設計者はさらに改善をしたり，最終的に実現する場合に，ずっとうまく他人に仕事を割当てることができるようになる．

1.4.3　機構の少数化

設計と保守のために，可能な限り数少ない機構でシステムの大部分の仕事をさせることが重要になる．たとえば，システム中にはリソースを使う要求の待ち行列が何ケ所にも存在する．このような待ち行列は，統一的な機構で取り扱うことができる．プロセスや機器の状態を記録するテーブルがシステム中に多数存在するが，これらのテーブルはアクセスや誤り検査などの面からみて統一的な方法で取り扱うことができる．

1.4.4 設計に対する理解

中心となる設計者が少数のときは，設計がモジュール化され，最少数の基本的機構を使って進められていることを保証するのは最も簡単なように思われる[23]．事実，システムが主モジュールとサブモジュール群の一般的機能や相互接続が理解できない程複雑な場合は1人の人では設計を制御しきれなくなる危険性が出てくる．

また経験では，設計が批評され，文書化されるまで製作を始めないことは良いと言われている[23]．管理者および設計者はプログラム製作に対してたびたび短気をおこすが，システム全体の開発を早める最良の方法は，微妙な設計段階ではゆっくり進むことである．

一般に，所定の機能を達成する方法はいくつかある．秀れた設計者は，それらの異なった可能性を考え，選んだ特定の方法については説明ができるようになっているのが普通である．もし別の方法が考られなかったり，選択した方法の説明が明確にできなければ，その問題を完全に理解してないことになる．実現する前に設計を顧みたり，文書化しなければ，難しい問題を省いたり，理解が欠けていることを見落としがちになる．文書はシステムを保守しようとする人々の技術レベルを持った人々が明確に理解でき得るものでなければならない．設計者はもとより，製作者，保守員，そしてユーザに至るまでの多くのレベルの人々がその仕様書を使う必要がある．設計が明確に説明されていなければ，おそらくその仕様書は理解されないだろう．

1.4.5 仕様決定の過程

さらに詳しく仕様を決める過程の方法を述べるに際し，Trapnell[25] が書いた方法と議論を対比する．これは大変有益なことである．われわれの観点では，設計と仕様決定の過程は同一のものである．その過程は反復的なものであり，個々のモジュールの一般的な必要条件から始まり，詳細な仕様に進む．機能上の仕様を高いレベルで扱うときに，多くの新しいサブレベルの問題が明らかになってくる．各段階において高レベルの機能の可能性を決定するために，より詳細な設計のレベルに進まざる得ないが，詳細なレベルには一時的にだけとどまることに注意しなければならない．この時点でサブレベルの問題を詳しく解くべきではない．すぐに詳細なレベルに移ってしまって，そこに固執している設計者を見かけるが，それは多くの重要で一般的な考察を見逃すだけである．

仕様を決定する手順は，次に述べる一般的なレベルからできている：

1. 外部仕様．このレベルはユーザの使える機能およびユーザとシステム間のコミュニケーションに要求される機能を明確にする．
2. 外部の機能を備えるために必要な主な内部機能ユニットの決定．
3. 機能相互間のコミュニケーションの規約と機構の仕様．このレベルはリンケージや各種テーブル等の仕様を決める．必要な機構数を調整できるのはこの段階である．
4. プログラム・モジュールの仕様．このレベルは各プログラムが要求するすべての入力，プログラムが作り出すすべての出力，およびテーブルのようなシステム的なものをすべて決める．
5. 詳細なプログラムのコーディング．

上で述べたレベルのすべてに関係することは，性能と大きさの要求，および一連のテスト手順である．各レベルに対応したテスト方法の設計は難しく手数がかかるが，すべてのサブモジュールを最終システムに容易に統合しようとするならば，実際の製作と同程度の高い優先度を与えなければならない[25]．

仕様書は，システムがいかに動作し，その世界とコミュニケーションするかといった一般的なシステム解説から始まり，階層的に構成すべきである．すなわち，システム設備や主な機能モジュールについて解説し，各機能および副機能から個々のルーチンに至るまで，設備やコミュニケーション方法に関する一般的な解説を載せるようにする．

設計の課程を通して，設計者は当然注意すべきすべての状況と条件を考えたかどうかを自問しなければならない．あるレベルの文書が満足できた時に，次のレベルの詳しい考察に移ることができる．主な機能の仕様を決める際に，しばしば並列的に仕事を進めることがあるが，一般的なことから特定なことに進めるという直列的な方針を全体として維持できるに違いない．低いレベルの機能を考えることは高いレベルの仕様で必要となった変更を見出せない．したがって，全体的なシステム設計を通して繰り返しの手法が必要である．

あるプログラマにとっては，この種の注意深い方法はやっかいなものであろうが，もし効果的な設計を進め，実現し，テストすることにすれば，ある種の訓練が必要と思われる[23]．仕様決定での過程で計算機の助けを借りて，モジュール間の接続，テーブルの使用法等をプリントアウトすることができる．このように計算機による手助けは，モジュールやシステム設備を変更する際に，変更の必要性があるかどうかを決定するのに役立つ．ハードウェア設計者は，こ

れらを部品リストや配線リスト等に利用できる．ソフトウェア設計者もまた計算機の助けを借りて良い効果をあげることができる．

製作が進むにつれて，あるモジュールの性能が仕様通りにならないことがある．プログラマはシステムのやりとりの約束に従わず，関連するモジュールにより直接的にコミュニケーションすることによって，いかに性能を改善できるかを知ることができる．しかしながら，このような近道はできるだけ避けてその代わりにシステムのコミュニケーション規定を再考察すべきである．

大規模なシステムの設計，実現には技術的な熟練以外のものが要求される．たとえば，組織や人を取扱う管理上の熟練が重要になる．

1.5 プロセスの概念

1.5.1 仮想プロセッサとプロセス

本書を通して重要な概念である**仮想プロセッサ**（virtual processor）とそれに関連した**プロセス**（process）について，ここでさらに詳しく調べる[11,27,31]．

われわれの良く知っているタイムシェアリング・システム設計者のどのグループも，この用語に違った意味付けをしているように思われる．たとえば，あるグループでは，プロセッサという用語を物理的なハードウェアのプロセッサに限定しようとしているが，プロセスの概念は広く使っている[9,27]．他のグループは，仮想プロセッサの概念を**制御プログラム**（control program）によって生成される実体を論議する際に使うが，プロセスという用語はめったに使わない[30]．さらに他のグループは，両方の用語を使い，時には同じ意味に使っている[3,11]．どのグループも基本的には1組のハードウェアを多くの機構やユーザが共用し，各々のユーザがそのシステムの制御を独占しているように思わせる概念や機構を表わすための用語を作り出そうとしている．

プロセッサとプロセスの概念は，特定のシステムとの関係において厳密に定義できる．しかしながら，そのような定義をすることはその概念の多くの巾広く直観的な意味を取り除くことになるだろう．そこで本書では仮想プロセッサとプロセスの概念は直観的にし，たとえ以下の文章が厳密な定義の形をとっているにせよ，厳密には定義しないようにする．以下に続く議論は，その概念の直観的な理解のために付け加える．

プロセッサは情報を変形する実体である．さらに簡単に言えば，プロセッサはコマンドや命令を実行する．プロセッサは，(1) オペレーティング・システ

ム，コンパイラ，コマンド言語翻訳ルーチンのようにソフトウェアで作ることができ，あるいは，(2) 演算論理装置（中央処理装置）やI/Oプロセッサのようにハードウェアで作ることができる．また，(3) 本書で使っているように，ハードウェアとソフトウェアの機能の組合わせとして作ることもできる．**プロセス**は仮想プロセッサを制御し，定義することができる実体である．ここで興味深いことは，オペレーティング・システムが任意の数の仮想プロセッサを生成するということである．これらの仮想プロセッサの1つを定義し，制御できる実体がプロセスである．**プログラム**はプロセッサに対する命令の順序を示すものであるが，プロセスは1つあるいはそれ以上のプログラムから成る実体であり，これらのプログラムを実行する仮想プロセッサを定義する付加的な情報を含むものである．この特別な情報の性質は以下に論じる．

仮想プロセッサとそれらを定義するプロセスとの間には1対1の関係があるから，仮想プロセッサとプロセスという用語が互いに同じ意味にとられることもあり，また，しばしばそれらの一方を冗長として扱うことがある．本書では互いに同じ意味をもつように解釈したい．これらの用語についてはさらに以下で論じる．

1.5.2 アドレス空間

アドレス空間の概念は，プログラマにとってシステムの記憶管理とアドレス方式の問題に関係するため明確にしておく必要がある．**アドレス方式**(adressing) はプロセスがそのアドレス空間の記憶場所を区別する手段である．実際に2つのアドレス空間があるが，それらに非公式な定義をしよう：

1. **物理アドレス空間** (physical-address space)：直接アドレス可能な実際の主記憶場所の集合からできている．
2. **抽象** (abstract)，**仮想**(virtual) あるいは**論理アドレス空間**(logical-address space)（しばしば，**ネーム・スペース** (name space) と呼ばれる)：プロセスによってアドレスされる仮想的あるいは論理的記憶場所の集合からできている．

論理アドレス空間は論理的記憶場所の1次元配列になっているか，あるいは，もっと複雑な構造である．論理アドレス空間では，記憶場所は何らかの方法で番号付けされたものとして考えられる．論理的な記憶場所は各々番号で識別され，その番号がアドレスである．ハードウェア・プロセッサ上で実行されるためには，少なくともプロセスのいくつかの命令とデータが物理的な主記憶

内に存在していなければならない．命令やデータが物理的な記憶装置内に存在するときに，適当な論理的な空間を用意し，論理アドレス参照をそれらが存在する物理アドレスに変換する機構を設けることが設計上の問題になる．もし論理アドレス空間が主記憶の物理アドレス空間よりも大きいときは，**仮想記憶**（virtual memory）という用語がしばしば使われる．本書では，仮想記憶，仮想アドレス空間，論理記憶，そして論理アドレス空間といった用語は，同一の意味に用いている．

仮想プロセッサが使うことのできる論理，あるいは，仮想アドレス空間は，ハードウェア・プロセッサが使える物理アドレス空間よりも大きいこともあるし，同じ場合もあるし，小さいこともある．主記憶は補助記憶（ドラムあるいはディスク）と一緒にすると，システム内のすべての仮想プロセッサに対する仮想アドレス空間の全体を収容する．プロセッサの制御下にあるプロセスに関連した全仮想空間の一部を主記憶内に維持するのはシステムの仕事である．論理アドレス空間から物理アドレスへの変換，すなわち**マッピング**（mapping）は，ハードウェアとソフトウェアの双方で取り扱われる．論理アドレス空間を生成し，論理アドレスを物理アドレスに変換するいくつかの常識的な技術については，第2章で論じる．

1.5.3 文脈ブロック

1.3.5 節で述べたように，各仮想プロセッサが命令セット，アドレス空間，補助記憶へのアクセス，他のプロセッサおよび種々のI/O機器とコミュニケーションする能力，自分自身を終結する能力をもっているものと解釈すると便利である．特定のプロセッサを定義するために必要な情報は，テーブルやデータ語のセットに含まれている．この情報の集まりは，しばしば1つにまとめられ，**文脈ブロック**（context block）と呼ばれる．1つの文脈ブロックとそれにより定義された仮想プロセッサで実行されるプログラムやデータは，1つのプロセスを構成する．

ここでは，文脈ブロック中に必要な情報のいくつかについて概略を述べる．物理プロセッサをあるプロセスから他のものへ切換えるためには，プロセスが制御からはずされるときに，ある情報を退避しなければならない．さらに，そのプロセスに制御がもどされるときには，再び元通りに回復しなければならない．次に述べるものは，退避および回復すべき情報の種類である[27]：

1. プロセスは物理プロセッサの制御を再び受け取るときに，次にどの命

令を実行するか知らねばならない．それゆえ，プログラム・カウンタは退避および回復しなければならない．

2. 物理プロセッサのレジスタ類の内容は，処理が始まる前に回復できるように退避される．

3. プロセスのアドレス空間は退避しなければならない．すなわち，システムはプロセスの仮想記憶の内容が主記憶内か，補助記憶内か，あるいはその2つの組み合せのどれに，そしてどこにあるかを知らなければならない．

4. そのプロセスに関係するか，使われている実際のI/O機器，あるいはシミュレーションされているI/O機器の状態は退避される．

仮想プロセッサを定義するには，さらに次のような付加情報が必要である．たとえば，アクセス可能なI/O機器の数とその形式，使用可能な記憶容量，仮想プロセッサに与えられるその他の機器である．プロセスが物理プロセッサの制御下にないときは，その文脈ブロックを補助記憶内に置くことができ，その文脈ブロックのポインタとスケジューリングの目的に必要な最少の付加情報だけを常に主記憶内に残しておけば良い．言い換えれば，文脈ブロックはそのプロセッサの内部レジスタ類のイメージを含む仮想計算機システムを表現している．

プロセッサの実行を開始するためにハードウェア・プロセッサおよび主記憶内に置かなければならない文脈ブロックの最小部分は**状態語**（state word），あるいは**状態ベクトル**（state vector）と呼ばれる[32,33]．状態ベクトルはオペランド，アドレス・レジスタ類の内容，論理アドレス空間から物理アドレス空間に変換するのを助けるレジスタ類の内容，および，プロセッサ内にセットされる付加状態や保護のための情報を含んでいる．

プロセッサをあるプロセスから他のプロセスへできるだけ敏速に切り換えるために，プロセッサ・レジスタに対するその他の要求事項に関連して状態ベクトルの大きさを最小にしたい．プロセスが実行されている間に，主記憶内に存在する文脈ブロックの付加情報は実際にそのプロセスを開始させるために状態情報が切り換えられる前に，プロセスのプログラムあるいはデータの全部あるいは一部とともに，主記憶内に持込まれる．場合によっては，必要になった時に持込まれる．補助記憶への文脈ブロック情報の転送は，時には他のプロセスの実行と同時に行なわれることが多い．

先に仮想プロセッサはその文脈ブロックにより定義されることを知った．プロセスを生成するためには，文脈ブロックを生成するだけで良い．さまざまな

出版物で仮想，論理，抽象空間あるいはネーム・スペースと呼ばれる仮想プロセッサのアドレス空間は，文脈ブロック中に格納されている情報により定義される．詳細は，第2章と第5章で論じる．

仮想プロセッサの命令セットは，物理プロセッサでの命令セットと同一の場合もあり，またそのサブセットであったり，それが拡張されている場合もある．I/O 命令のような物理プロセッサの命令をシステムがいかに取り扱うかによって，同一セットあるいはサブセットとなりうる．仮想プロセッサの命令セットには，I/O を扱い，他の仮想プロセッサに信号を送り，アドレス空間やファイル・システムを取扱うシステム・プログラムに対して呼出しを行なう機能を含めることができるので，命令セットは拡張されている．これらの呼出し機能は，物理ハードウェアよりもずっと簡単に利用できるプロセッサを生成し，多くの場合，それは豊富な機能を備えている．これらのシステム・プログラムを呼出す機能は重要な概念であり，システムの保護を適切に行ない，管理上の浪費を最小にするように注意して設計する必要がある．この話題については，第4章と第5章で論じる．

1.5.4 ユーザとジョブ

ここでいかにしてシステムが**ユーザ**（user）あるいは**ジョブ**（job）と呼ばれる実体にプロセスを割付け，かつ責任をもつかについて論じる．この節におけるいくつかの議論は Lampson のものに従っている[32]．最も一般的な形で，1つのユーザあるいはジョブは，いくつかのプロセスに関係をもっている．1つのジョブに1つ以上のプロセスを関係付けたいという理由はいくつかあるが，その内の3つは次のことである．(1)システム内に存在する多数のハードウェア・プロセッサの利点を利用するため，(2)1つのジョブ内に存在する同時処理の可能性を利用するため，そして，(3)システムの限界を避け，プログラム構成を単純化するために，ジョブを小さなユニットに分解できるようにするためである．プロセスと仮想プロセッサの間には1対1の関係があるので，同時に実行したり，プロセス群を独立化することが可能である．

1つのプロセスが同時処理が可能なところまで進むとシステムを呼出し，他のプロセスを生成する．通常，プログラマおよびシステムの双方にとって，1つのジョブのプロセスを階層的に**ツリー構造**（tree structure）として構成すれば都合が良い．プログラマにとっては，階層的構成は制御や他の情報の流れが単純な形になるので都合が良い．システムにとっては，ツリー構造に含まれる情報や制御の制限が，プロセス相互のコミュニケーションに必要な構造を単

純にしている．

ツリー構造において，与えられた結合点（node）で始まっているプロセス同志は互いに並列的であると言われ，互いに直接情報を交換することはない．すべてのコミュニケーションはツリーの枝にそって行なわれなければならない．1つのジョブ内のすべてのプロセス間で直接的なコミュニケーションを許すもっと一般的な構造の形式も可能で，システム設計において作ることもできる．しかしながら，話を簡単にするためにここではプロセスを生成するジョブの構造はツリー構造だけを考えるものとする．

ジョブの基本的な特徴は，システムに関する限り，他のジョブとは完全に独立なことである．一般に，ジョブ内のプロセスは互いに他と関係があり，コミュニケーションし，リソースを共用し，そして連動する．ツリーの根（root）に当たるプロセスは，システムによって生成され，そのプロセスはシステムに対するコマンドを認識し，下位レベルのプロセスの不当動作を処理し，他のプロセスを開始させたり，消滅させたりする役割を果たす．ジョブとそのプロセスのツリーを，図 1.2 に示す．そこではプロセス2がツリーの根なっている．プロセス21から23は，プロセス2によって生成されたもので，互いに直接コミュニケーションできない．それらを生成した親プロセス，あるいはそれらがすでに生成したプロセスを通してのみコミュニケーションができる．同様に，プロセス211と212はプロセス21によって生成されたもので，それらは並列的である．各プロセスはそれ自身の仮想プロセッサ上で実行される．プロセス同志は互いにコミュニケーションでき，記憶を共用できる．1つのジョブにいくつかのプロセスが存在する一般的な例としては，デバッグのためのプログラムの使用法がある．この場合，**ルート・プロセス**（root process）はデバッグ・プログラムを実行させる子のプロセスを生成する．デバッグ・プログラムは，続いてデバッグされるプログラムを実行させる子のプロセスを生成する．並列的なプロセスから出来ているツリーのより一般的な使い方は，プログラミング言語における並列動作を定義する機能およびプロセス相互間のコミュニケーションに対する機能に依存している．この問題は，さらに第5章で論じる．

課金計算は原理的には個々のプロセスに対して行なうことができるが，システム内では，通常，ジョブやユーザが基本的な課金計算の対象となる．一般に，ユーザあるいはジョブは，次のような属性を所有している[32]：

1. 名　前．
2. あるリソースを使用する権利．

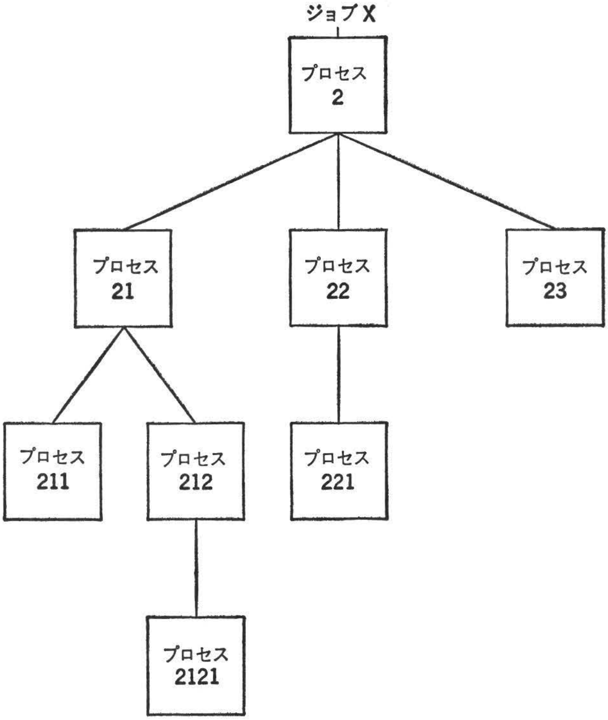


図 1.2　ジョブとそのプロセスのツリー

3.　プログラムやデータを保持する一時的および永久的なファイルの集まり，そして，おそらく他のユーザやジョブのファイルをアクセスする権利.
4.　いくつかのプロセスの集まり.

ユーザに関係するものは，ある種の保護と**アクセス権** (access right) である．ユーザは名前を付けられた課金計算の対象となる実体であり，したがってシステムをアクセスするこの権利を持てる何人かの人になり得る.

もしシステムをそのように設計していれば，適切な権利に従って，ジョブあるいはユーザが他のジョブのプログラムやデータをアクセスできるということを強調しておきたい．本書では，**ユーザ**という用語はしばしば端末にすわっている人のことを指すが，ユーザの概念は抽象的である．上で述べたユーザの性

質に留意しておく必要がある．

ユーザについての議論では，ユーザと遠隔端末との関係については触れなかった．多くのシステムでは，システムで**活動中**（active）のジョブとコンソールにすわっている人の間には，1対1の関係が存在する．その最も一般的な形では，ユーザまたはジョブの概念は活動中のジョブに含まれるが，端末に関係しないジョブは含めないこともある．たとえば，ユーザは**ログ・イン**(login)し（システムとの交信を確立し），プログラムを作り，そのプログラムのコンパイルやデバッグを行ない，さらにそのプログラムをファイルに入れることがある．それから計算機時間の安い夜間に他のファイルにあるデータを使って，そのプログラムを実行し，その結果をそのまま他のファイルに残すよう指示する．ユーザはその後**ログ・アウト**（log out）する(システムとの交信を断つ)．夜になって，システムは端末とは接続なしにジョブを生成し，それを実行する．

端末に関係しないプロセスを行なう他の例に，システム機能を実行することがある．たとえば，システムの負荷が軽いとき，または数秒あるいは数分毎に定期的に実行する場合に，予防的な保守や，システム・テーブルやデータ構造に矛盾がないかどうかをチェックしたり，I/O 機器が正常に動作しているかを調べるためのプロセスを実行させることがある．その他に最優先順位ではないが，一般のユーザの必要性と同程度にリソースを必要とするシステム機能を行なう場合がその例である．

1.5.5　ま　と　め

ここでの議論はシステム設計を通して，仮想プロセッサとその制御の対象物であるプロセスの概念の有用性を示してきた．いかにしてプロセスが生成されるか，そしてその命令セット，アドレス空間，およびコミュニケーションのどのような性質が必要であるかを示した．仮想プロセッサの割付け，ユーザと物理ハードウェア・モジュールの両方に関係した付加概念をも紹介してきた．プロセスの概念とその利用については，さらに第5章で論じる．

1.6　コンソールの使用

1.6.1　システムとの交信

ユーザの要望を満足するために必要なシステムの機能の形式は，XDS-940

のような典型的なシステムでの端末使用法にみることができる．この他にも別の概念が出てくるが，それは本書の後で詳しく論じる．

ユーザが端末にすわり，もし端末が直接システムに接続されていれば，スイッチをオンにし，端末が交換回線を介して接続されていれば，システムにダイアルすれば，システムはこの端末のために1つのジョブを生成する．そのジョブはいくつかのテーブルの割付けと初期化を行なうことによって作られ，**ジョブ番号**（job number）が与えられる．システムはこのジョブ番号を，ジョブを管理する際に使う．さらに端末から入力を受け取り，コミュニケーションするためのプロセスを生成する．このプロセスは通常，コマンド通訳プログラムで，それを本書では**ルート・プロセス**（root process）と呼ぶ．コマンド通訳プログラムはユーザに “please log in” を送って，ユーザの認識を要求する．

遠隔端末は名前が付いておらず，だれでも使え，電話でシステムと交信できるので，ユーザは適当な識別情報を送らねばならない．通常，この識別情報は課金コードおよびプロジェクトなどの名前である．システムはこの識別情報が正当なユーザのテーブルにあるかをチェックし，もしあれば，ユーザに次に進むよう指示する．この時点で，システムはテーブルに格納されている情報から，そのユーザが使えるアクセスの特権を知る．コンソール使用中に利用したシステム・リソースは後でユーザの課金を計算するために記録される．

1.6.2 プログラムの作成

ユーザはプログラムを書き，それをデバッグし，共同作業者のファイル中に格納されているデータに対して，そのプログラムを実行したい場合がある．そのためシステムに対し，テキスト・エディタと交信できるように指令する．システムはテキスト・エディタに対するルート・プロセスの新しい子プロセスを生成し，制御をエディタ・プロセスに移す．エディタ・プログラムはそれ自身のコマンド言語を持ち，それを使ってユーザはテキストを入力し，削除し，修正し，そしてファイルに書くことができる．エディタはファイルに名前をつけることを要求する．たとえば，PROG という名を付けたとすれば，この情報を扱う間中，この名前を使うことができる．もし望むならば，その名前を変えることもできる．コマンドにより明白に宣言しない限り，システムはそのファイルを**個人的**なものと仮定し，このユーザと同一の識別情報でログ・インした人のみがそれをアクセスできる．

ファイルを作成し終わると，ユーザはコンソール上のシステム呼出しボタン

を押す．エディタ・プロセスは終結され，制御はルート・プロセスにもどされる．その後，ユーザは適当なコンパイラを使ってファイルPROG中に格納されたソース・プログラムをコンパイルし，その2進出力をBINARYと名付けた他のファイルに格納するように指示する．ルート・プロセスはファイル・システムに尋ねて，コンパイラを制御する他のプロセスを生成し，コンパイラにファイルの物理アドレスを渡す．その後，コンパイラに制御を移す．

1.6.3　リエントラント・プログラム

本節を離れて，1つの重要な概念について簡単に触れておく．同じ時間にコンパイラやエディタを使っている人が何人もいる可能性がある．本質的には，このことは問題ない．なぜならば，各ユーザにこれらのプログラムの複写を別々に用意することができるからである．しかし，主記憶は貴重なリソースであり，それゆえ，特に主記憶内ではこれらのプログラムのただ1つの複写のみがシステム中にあるようにして，それを共用したいのが普通である．この要求はプログラムとプロセスを区別すべきことを示している．共用されているのはたった1つのプログラムであるが，各々のジョブにとってプログラムを別のプロセスの一部として使うということは，データやプログラムを別のアドレス空間に蓄えたり，あるリソース割付けルーチンの制御の下で，1つのプロセスから他のプロセスへハードウェア・リソースが切り換えられるときに，各ジョブ毎にプログラムの実行状態を覚えておくことが必要である．

ただ1つのプログラムで複数個のプロセスが同時に使うことのできるものを**純手続**（pure procedure）あるいは**リエントラント**（reentrant）であるという．リエントラントであるためには，プログラムはある条件を満足していなければならない．すなわち，(1)実行中にそれ自身がプログラムを変えてはならない．(2)それ自身の中にデータを格納してはならない．第1の条件は，コンパイラ・コード生成ルーチン内で命令を変更するようなプログラミング技術を使わなければ容易に達成できるし，アセンブリ言語のプログラミングで容易に達成できる．第2の条件は，データを分離して，そのプログラムのための一時的な記憶領域を設けることにより達成できる．一般にそれはプログラムを使う各プロセスのアドレス空間内にとられる．プログラムを共用するための技術について，第2章で述べる．さてここで本筋にもどろう．

1.6.4　プログラムのコンパイルとデバッグ

コンパイラは，ファイルPROGにあるプログラムをコンパイルし，ユーザ

のいるコンソール，あるいは，後の検査のための指定されたファイルに，文法誤まりを報告する．BINARY と名付けられたファイルに正しい2進プログラムが格納されるまで，ユーザは何回でもエディタの呼出し，プログラムの修正およびコンパイルを行なうことができる．

別のプログラム作成方法は，組込みの編集機能を持ちソース・コードの各行が入力される毎に1行ずつコンパイルする会話形コンパイラを使うことである．この形式のシステムは，ユーザに1行毎の実行を許し，新しい行の挿入も可能にしている．

ユーザは，2進ファイルを作成し終わると，論理的な誤りをデバッグする場合がある．この場合は，ルート・プロセスに対してデバッグ・プログラムを提供するように指示する．ルート・プロセスは，デバッグ・プログラムを実行させるプロセスを生成し，それを制御する子プロセスを生成する．これら2つのプロセスはデバッグの間中，制御を移したり戻したりしてコミュニケーションする．この例は，プロセス間でコミュニケーションが必要である場合を示している．

1.6.5 プログラムの実行

デバッグが終了すると両方のプロセスは終結し，ユーザはファイル BINARY 上のプログラムを DATA という共同作業者のファイルからのデータを使って実行し，その結果を端末に表示するように要求する．ファイル・システムはこのユーザがファイル DATA のアクセスを許しているかどうかをチェックし，もしそうであれば，このことをルート・プロセスに報告する．そして，ルート・プロセスはさらにそのプログラムとデータを実行する別のプロセスを生成する．入力および出力ファイルのためのファイルの名前付けがユーザのプログラム内で行なわれている場合は，実行時にチェックが行なわれる．そのユーザのプログラムが実行されるが，（それがどう書かれているかによるが）実行プロセスはユーザに次の入力を要求したり，あるいは結果を報告したり，場合によってはコンソールから実行を続行させる信号を受け取るまで休止したりする．ユーザは仕事が終わると，ルート・プロセスと再び交信してログ・アウトする．コンソールを使っている間に作成されたファイル類は，次の作業のためにとっておかれる．

1.6.6 コンソール使用中のシステム機能

ユーザには見えないが，コンソールを使用中，システムは多くの機能を実行

する．ユーザのファイルを磁気テープや他の記憶媒体上に複写するためにファイル・バックアップ・プログラムを周期的に使ったり，システム状態をチェックするために診断ルーチンを実行することがある．そして，リソース管理ルーチンはプロセスを生成し，割付け，消滅させたりして，それぞれのユーザに満足行く応答を与えるように，プロセスからプロセスにハードウェア・リソースを切り換える．低い優先順位のバッチ・モードでも，ジョブを実行するためにいくつかのプロセスが生成される．補助記憶，I/O 機器および主記憶間の情報をバッファリングしたり転送するルーチン類が実行される．システムはどんな情報が各々のジョブに属しているかを記憶し，他人からユーザの機密と保護を保証している．また，リソースの利用に関しても公正かつ適切な方法で計算する．

1.7 まとめ

この章は本書の構成を述べたものであり，いくつかの重要な概念や要求事項，および後に続く章でさらに詳しく論じる問題点を紹介した．ここで紹介した基本となる考えは，汎用タイムシェアリング・システムの考えを含んでおり，また本質的問題点と技術問題点の区別を行なった．ユーザに割付けるために，ハードウェア・リソースそのものを仮想プロセッサやファイル・システムにいかに構成するかを知った．オペレーティング・システムがこの機能を果たした後に実際の仕事を遂行するためには，プロセスからプロセスへハードウェア・リソースの制御を切り換えなければならないことも知った．全体の動作の中には，おそらく保護やバックアップ機構が含まれ，課金計算や保守，評価のための計測が行なわれるだろう．

われわれは後に続く章の中で議論されるものに対しては，大ざっぱに扱ってきた．時には，本書を通して使われる中核となる概念については，いくらか詳しく述べてきた．後に述べることは，2つに大別することができる．すなわち，ハードウァアとソフトウェアの概念である．すでに指摘したように，これらの概念を明確に分割することはできないし，時にはそれらは重なっている．概要を述べた基本的な問題の多くはいつまでも残るであろうが，特定の解における実現方法は変わっていくであろう．ハードウェアとソフトウェア間には密接な関係があるため，ソフトウェアの概念のあるものはハードウェアの章で論じるが，それはその論議の適当な下地を作るためである．また，ハードウェア概念のあるものは，ソフトウェアで議論した方がずっと自然であるように

思える．

参 考 文 献

1 **Corbato, F. J., et al.:** An Experimental Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 21, pp. 335–344, 1962.

2 **Schwartz, J., E. G. Coffman, and C. Weissman:** A General Purpose Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 25, pp. 397–411, 1964.

3 **Corbato, F. J., and V. A. Vyssotsky:** Introduction and Overview of the Multics System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 185–196, 1965.

4 **Sackman, H.:** Time-sharing versus Batch Processing: The Experimental Evidence, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, pp. 1–10, 1968.

5 **Gold, M. M.:** Time-sharing and Batch Processing: An Experimental Comparison of Their Values in a Problem Solving Situation, *Commun. ACM*, vol. 12, no. 5, pp. 249–259, May, 1969.

6 **Corbato, F. J., and J. H. Saltzer:** Some Considerations of Supervisor Program Design for Multiplexed Computer Systems, *IFIP Conf. Proc., Edinburgh, August, 1968.*

7 **Evans, G. J., Jr.:** Experience Gained from the American Airline SABRE System Control Program, *Proc. ACM Natl. Conf., 1967*, pp. 77–84.

8 **Shaw, J. C.:** Joss: A Designer's View of an Experimental On-line Computing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 26, p. 455, 1965.

9 **Lichtenberger, W. W., and M. W. Pirtle:** A Facility for Experimentation in Man-Machine Interaction, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 589–598, 1965.

10 *IBM Systems J.*, vol. 5, no. 1, 1966 (devoted to a description of OS/360).

11 **Saltzer, J. H.:** Traffic Control in a Multiplexed Computer System, MAC-TR-30, thesis, Massachusetts Institute of Technology, Cambridge, Mass., July, 1966.

12 **Nielsen, N. R.:** Computer Simulation of Computer System Performance, *Proc. ACM Natl. Conf., 1967*, pp. 581–590.

13 **Nielsen, N. R.:** The Simulation of Time-sharing Systems, *Commun. ACM*, vol. 10, no. 7, pp. 397–412, July, 1967.

14 **Bryan, G. E.:** JOSS 20,000 Hours at the Console: A Statistical Survey, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 769–778, 1967.

15 **Calingaert, Peter:** System Performance Evaluation: Survey and Appraisal, *Commun. ACM*, vol. 10, no. 1, pp. 12–18, January, 1967.

16 **Campbell, D. J., and W. J. Heffner:** Measurement and Analysis of Large Operating Systems during System Development, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pp. 903–914, 1968.

17 Cantrell, H. N., and A. L. Ellison: Multiprogramming System Performance Measurement and Analysis, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, pp. 213–221, 1968.

18 Estrin, G., et al.: SNUPER COMPUTER: A Computer in Instrumentation Automation, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 645–656, 1967.

19 Freeman, David N., and Robert R. Pearson: Efficiency vs. Responsiveness in a Multiple-services Computer Facility, *Proc. ACM Natl. Conf. 1968*, pp. 25–34B.

20 Schulman, Franklin D.: Hardware Measurement Device for IBM System/360 Time-sharing Evaluation, *Proc. ACM Natl. Conf. 1967*, pp. 103–109.

21 Sherr, A. L.: An Analysis of Time-shared Computer Systems, Ph.D. thesis, *MIT Project MAC Doc.* MAC-TR-18, June, 1965.

22 Varian, L. C., and E. G. Coffman: An Empirical Study of the Behavior of Programs in a Paging Environment, *Commun. ACM*, vol. 11, no. 5, May, 1968.

23 Corbato, F. J.: Sensitive Issues in the Design of Multi-use Systems, *MIT Project MAC Mem.* MAC-M-383, Dec. 12, 1968.

24 Kay, R. H.: Management and Organization of Large Scale Software Development Projects, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 425–433, 1969.

25 Trapnell, F. M.: A Systematic Approach to the Development of System Programs, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 411–418, 1969.

26 Madnick, S. E.: A Modular Approach to File System Design, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 1–13, 1969.

27 Lampson, B. W.: A Scheduling Philosophy for Multi-processing Systems, *Commun. ACM*, vol. 11, no. 5, May, 1968.

28 Rosen, Saul: Electronic Computers: A Historical Survey, *ACM Computing Surv.*, vol. 1, no. 1, pp. 7–36, March, 1969.

29 Rosin, Robert F.: Supervisory and Monitor Systems, *ACM Computing Surv.*, vol. 1, no. 1, pp. 37, 54, March, 1969.

30 CP/CMS Program Logic Manual, IBM Cambridge Scientific Center, Cambridge, Mass., 1969.

31 Dennis, J. B., and E. C. Van Horn: Programming Semantics for Multiprogrammed Computations, *Commun. ACM*, vol. 9, no. 3, pp. 143–155, March, 1966.

32 Lampson, B. W.: Scheduling and Protection on Interactive Multi-processor Systems, *Univ. Calif. Berkeley Proj. Genie Doc.*, 40.10.150, Jan. 20, 1967.

33 Dennis, J. B.: Segmentation and the Design of Multiprogrammed Computer Systems, *J. ACM*, vol. 12, no. 4, pp. 589–602, October, 1965.

第 II 部

ハードウェアの概念

第 2 章

記憶装置，アドレス方式，および割付け

2.1 概　　要

2.1.1 リソース共用システムにおけるハードウェアへの要求

汎用タイムシェアリング・システムに要求されるハードウェア機能は，1.2節で紹介した他の目的のタイムシェアリング・システムにも要求される．そのようなシステムの基本的な特性はシステム・リソースの動的な割付けである．そのようなシステムによって要求される主なハードウェアの機能は，(1)プロセスの他のプロセスからの保護，ならびにシステムのシステム自身およびユーザ・プロセスからの保護を助けるための保護機構，と(2)リソースの動的再配置を効果的に行なうための機構，である．これらの要求事項は互いに関連がある．このようなシステムにさらに要求されるものは高信頼性である．保護および制御機構については第4章で論じる．信頼性に関しては第7章で論じる．本章では主記憶の割付けを助けるための機構について述べる．第3章ではリソース共用システムで生じる重要なハードウェア・コミュニケーションの問題について論じる．

2.1.2 記憶システムの設計問題

現在のシステムでは主記憶がリソースの中核であると言える．主記憶は論理演算装置（CPU）や I/O プロセッサ（IOP）のために命令を蓄え，通信回線に乗せ，種々の I/O と 2 次記憶機器の間に転送する情報をバッファするために用いられる．また，常駐のオペレーティング・システムのためのプログラムを蓄えている．大規模なタイムシェアリング・システムを成功させるためには，記憶システムを的確に設計することが肝要である．過去数年前までは，図2.1 のモデルが標準の計算機システムであった．このシステムでは，演算処理

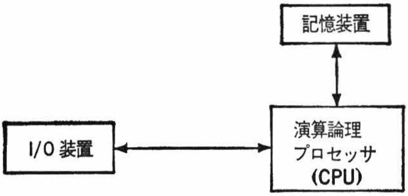


図 2.1　CPU が中核である計算機システムのモデル

装置が中央にあり，それを中央処理装置（CPU）と呼んでいた．ところが現在では，図 2.2 に示すような記憶装置が中核のリソースに見えるようなモデルになっている．

記憶システム全体の設計においては 5 つの主要な問題があることが分かる：

1. 記憶アドレス方式に関する技術の開発．これはプロセスが主記憶の物理的な空間よりも大きいような論理アドレスの空間をアドレスすることを可能にする．
2. 物理的記憶装置の割付けに関する技術，および論理アドレス空間を物理アドレス空間にマッピングするための技術の開発．
3. 記憶保護に関する技術の開発．
4. 記憶装置の構成およびアドレス，データ，ならびに制御線の構造（バス構造）に関する開発．これによって主記憶を使うすべてのプロセッサおよび機器が最小限度の干渉で最大速度で動作できるようにする．
5. 記憶装置の階層を装置の種類に無関係になるように使うための技術の開発．

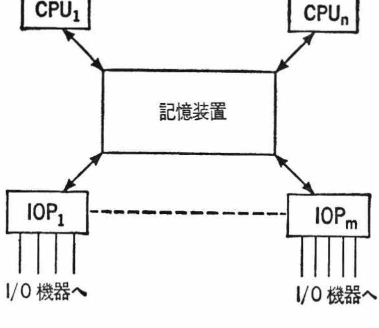


図 2.2　記憶装置が中核である計算機システムのモデル

この最後の問題は第6章で論じる．問題1から3は，通常，関連した技術で解決されるが，いままでにもかなりの注意が払われてきた．問題1と2は本章の主題である．問題3は第4章で論じ，問題4は最近になって初めて注意されるようになった．これに関しては，第3章で論じる．

2.1.3 アドレス方式および割付けの問題

タイムシェアリング計算機におけるアドレス方式と記憶割付法を適切に述べるためには，まず主記憶装置群の多重化した性質を十分に考慮しなければならない．原理的には，1つのプロセスに対して主記憶の一部分を割付けることはできるが，プロセッサに対してはそういうわけにはいかない．プロセッサは1ユニット単位のみ割付けることができる．しかしながら，プロセッサは容易に多重化することができるが主記憶はできない．なぜならば，主記憶と補助記憶の間で情報転送するための時間がかかるからである．主記憶を多重化するために主記憶と補助記憶の間でプロセスを移転させることを**スワッピング**(swapping）と呼ぶ．タイムシェアリング計算機のアドレス方式の設計において要求されるものの1つは，記憶装置の割付けの利点を強調し，多重化の不利益を最小限度にすることである．

複数個のプロセスが使っている主記憶においては特定のプログラム，たとえばコンパイラ等はたった1つの写しを持っている方が望ましく，各プロセスが別々の写しを持つようにするのは望まくしない．複数個のプロセスによって共用されるように設計されたプログラムの概念は，1.6.3節で述べた．このようなプログラムは**リエントラント・プログラム**あるいは**純手続**と呼ばれる．リエントラント・プログラムは2つの特性を持っていることを思い出そう．すなわち，(1)そのプログラムの命令またはアドレスがプログラムの実行中に変更されない．(2)一時記憶およびデータの領域はそのプログラム自身の外側に確保される．その領域は一般にリエントラント・プログラムを呼出しているプログラムの記憶領域にとられる．種々のアドレス方式を備えた計算機では，上述の機能をもったプログラムを書くことができるが，ある種のアドレス方式を使うとこのようなプログラムの記述および保護はずっと簡単になる．記憶装置をより効果的に使う他の方法は，プロセスが物理的な記憶装置のどこにでも置けるような融通性を持たせることである．物理的記憶装置内にプロセスを動的に再配置するこの機能は，種々のアドレス方式と記憶領域割付けの技術によって実現できる．

アドレス方式と記憶領域割付け法のユーザに対する影響もまた考えられなければならない．応用システムの設計および実現の費用－そしてまたある種の問

題にとりかかる可能性は，アドレス方式と記憶領域割付け法にかなり影響するように思われる．アドレス方式と記憶領域割付けシステムの設計における種種の決断には，ユーザの要求と共にシステム的な配慮が必要である．

設計者がしなければならない決断の1つは，論理アドレス空間の大きさを決めることである．特に物理アドレス空間に比べて小さくすべきか，同じにすべきか，あるいは大きくすべきかである．また，物理アドレス空間の構造も決定する必要がある．たとえば，良く使われている大きな線形アレー，Multics でみられるようなリンク可能な線形アレーを結合したもの，あるいはツリー構造などがある．この構造のどの程度をハードウェアで行ない，どの程度をソフトウェアで行なうかを決めなければならない．論理アドレスを物理アドレスに翻訳する，すなわちマッピングの技術も決定しなければならない．現在のシステムがこのマッピングを行なう場合は，次の3つの時点がある[1,2]：

1. **プログラムをそのまま実行可能なプログラムとして準備するとき：**その結果は**絶対プログラム**（absolute program）で，したがってそのプログラムを実行させる時はいつも同じリソースを割当てる．
2. **プログラムをロードするとき：**これは**静的再配置**(static relocation)として知られている．
3. **プログラムを実行するとき：**これは**動的再配置**（dynamic relocation）と呼ばれている，

上述の方法のいずれをとっても容易に記憶保護の方法は考えつくが，この話題に関する検討は，4.1.2 節で行なう．本節の以下では，論理アドレス空間の構造およびマッピング法の選択に影響を与えるいくつかの要因について考えてみる．ここでは，ハードウェアの記憶装置の構造として，線形アレーまたはその集合のみを考える．なぜならば，ツリー，リスト，またはリンク等のより特殊化された構造は，一般に，ソフトウェア・プロセッサによって実現されるからである．以下に続く再配置に関する節では，常識的に使われている技術を述べ，各技術が開発された動機について論じる．

2.2　静的再配置

データの参照を物理アドレスに翻訳することをプログラムの準備中に簡単に行なうことができるが，プログラムを共用したり変更したい場合にいくつかの困難な問題が起こる．たとえば，プログラムに1命令を挿入する場合，その命

令の後にある命令やデータに対するすべての参照を更新しなければならない．同様にすでに準備したルーチンの外にプログラムを独立に作る場合は，そのルーチンがロードされた場所を示すようにアドレスの参照を変更しなければならない．さらにまた，この場合の変更は，論理アドレス空間を物理アドレス空間の大きさに制限することになる．

静的再配置の処理は，かなりの計算時間が必要である．静的再配置を使っているシステムでは，一般にプログラムは最初の命令が0番地からロードされるかの如くアセンブルされる．そして続く命令とデータは，その点から連続したセルに逐次配置される．プログラムの最初の語の記憶場所を**ベース・アドレス**（base address）と呼ぶ．アドレス参照をもつ命令およびデータ語はすべてアセンブラが印をつける．そこで，ロード時に**ローダ**（loader）と呼ばれるプログラムが，すべてのアドレス参照がそのプログラムのロードされた実際のベース・アドレスに対してうまく合うように調整する．もし独立的にアセンブルされたいくつかのプログラムを1つにまとめてロードする場合は，ローダにアセンブラが与える情報を使って，プログラム相互のアドレス参照がすべて各々のプログラムの実際のアドレスを指すようにする．この処理を**リンキング**（linking）と呼ぶ．

静的再配置では，ユーザは記憶装置のどこにでもロードすることができる．しかしながら，プロセスが補助記憶に移され，スワッピング中に戻される場合は，ロードの処理を省くためには同じ場所に配置されねばならない．（ここで再びロードの処理を行なっても，そのプログラムは純手続の部分とデータ部分に分けなければならないし，そのデータ部分は絶対記憶アドレスを含まないようにしなければならない．）静的再配置の主な利点は，ロードをしている間に独立に書いたプログラムやデータを，適当にリンクした1つの計算にまとめることができることである．物理アドレス空間へのマッピングはローダによって適当に行なわれる．各プログラムは，それ自身の論理空間で書ける．しかしながら，同じ記号を重複して使うことは許されない．このような重複を許すようなプログラミングの手法を開発することは可能である．

プログラムを物理記憶装置内のどこにでも配置する機能は，リンクの処理には有用であるが，タイムシェアリング・システムにおける記憶装置の利用を効果的にするためにはあまり役立たない．たとえば，新しいプロセスを始める場合，システムは利用できるセルの集まりに適合するようなプロセスを見つけ出そうとする．仮に，このようなプロセスが見つかり，処理が終了するまで主記憶にプロセスが停まることができるならば，静的再配置でもいくつかのプロセ

スに主記憶を共用させることができる．（ある種の記憶保護方式を仮定することが必要である．この話題は別に検討する．）自由に使えるセルの数が，新しいプロセスが要求する数よりも多い場合は，よりありふれた場合であるが，そのようなセルは連続したブロックの中にはないのが普通である．

たとえ最初のロード時に連続したブロックが得られたとしても，スワッピングが要求されると，そのプロセスが実行されるたびにある情報を主記憶内の他の場所に移したり，補助記憶に移さない限り，同じ連続したブロックが必ずしも得られない．この理由から動的再配置ハードウェアがないシステムでは，タイムシェアリング・システムに使った場合，一般にたった1つのプロセスのみに一定時間，主記憶にはいっているようにしている[3]．したがって，スワッピング操作中はシステムは休止状態になる．動的再配置の方法が開発された動機はこの理由からである．

2.3　ベース・レジスタによる動的再配置

2.3.1　概　　要

動的再配置方式のうち最も簡単でかつ一般的な方式の1つは**ベース・レジスタ**[1] (base register) を使うことである．ベース・レジスタは記憶装置への操作があるたびに，内容を加算することのできるレジスタである．すべてのアドレスにベース・レジスタの内容を加えることにより，記憶装置のいかなる場所にも連続したセルのブロック単位にプログラムをロードすることができ，ベース・レジスタにはプログラムの適当なベース・アドレスをセットすることができる．ベース・レジスタを使えば，最初は静的再配置装置を使ってプログラムをロードするが，その後はロード処理をせずに，1単位として動的に再配置することができる．プログラムは論理空間に対してロードされるのであって，物理空間に対してロードされるのではないので，このような融通性が生まれる．ベース・レジスタは，論理空間を物理空間にマッピングするハードウェアのマップを構成する．ベース・レジスタが複数個あると，さらに融通性が得られる．その結果，プログラムを共用したり，非連続の記憶領域にプログラムを分割してロードすることが可能になる．

ベース・レジスタ方式には種々の変形がある．実際に，セグメンテーション（後述する）方式はいくつかのベース・レジスタと呼ばれるハードウェアを使って実現される．本章では上に述べて，図2.3に図示したような最も単純な形

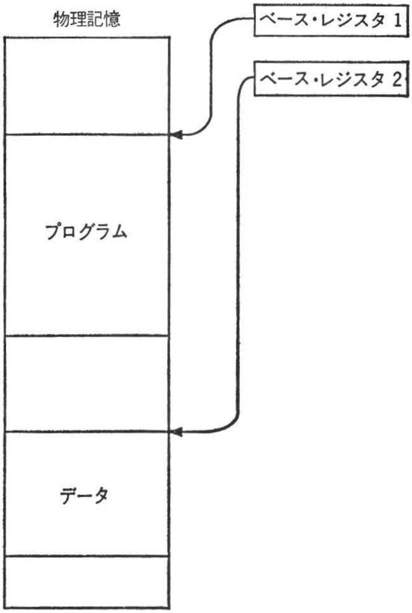


図 2.3 プログラムとデータの動的再配置に使われる2つのベース・レジスタ

式のベース・レジスタの概念を話題にしよう．アドレスを作る際にどのベース・レジスタを使うかを指定するのに2つの一般的方法がある．1つの方式は，IBM/360 によって代表されるが，プログラムが直接アドレスできるベース・レジスタが必要で，プログラムがベース・レジスタをアクセスできるようになっている．2番目の方式は UNIVAC 1108 に代表されるが，プログラムにはベース・レジスタのアクセスは許されず，実行される記憶装置の操作に応じて暗黙のうちにベース・レジスタをアクセスするようになっている．たとえば，命令取出しではいつも1つのベース・レジスタを使い，データの取出しと書込みにはいつも他のベース・レジスタを使う．動的再配置はどちらの方式を使ってもできる．しかしながら，Corbato が指摘するように，動的再配置を達成するためにはシステムにおいてプログラミング上の約束を必要とし，プログラムにベース・レジスタのアクセスを許すのが難かしくなる[24]．

プログラムの共用は，システムにおいて次のようにベース・レジスタを使い，リエントラント・プログラムを書くことによって行なわれる．すなわち，リエントラント・プログラムへの記憶参照は1つのベース・レジスタを使い，リエントラント・プログラムを呼出すプロセスにおけるデータへの記憶参照は別のベース・レジスタを使う．

2.3.2　論理空間の大きさ

ベース・レジスタによる静的再配置ないしは動的再配置の論理アドレス空間の大きさは，一般に，物理アドレス空間の大きさに等しいか，あるいは小さいのが普通である．ユーザはすぐに必要としない計算の一部に補助記憶から取り出した他の部分を承知の上で重ね書きすることによって，物理空間の方が大きく見えるようにすることができる．この処理は**オーバーレィ**（overlay）と呼ばれる．オーバーレィはスワッピングの概念と密接な関係があるが，その相違は前者がユーザの責任において行なわれるのに対し，後者はシステムの責任において行なわれることである．オーバーレィは主記憶の大きさの制限を克服するために個人的に行なわれ，スワッピングは同じ目的でシステムが行なう．オーバーレィを行なうには，プログラマは計算で使う物理記憶を注意深く構成する必要がある．注意深く計画をたて，同時に用いられるプログラムやデータ構造が論理あるいは物理空間内で同一場所を占めないことを保証する必要がある．オーバーレィを考えることを雑用と見るか，あるいはプログラミングの練習の好機と見なすかは検討してほしい．オーバーレィを考えたり，実現することを容易にするシステムの助けをプログラマに提供することは可能である．

2.3.3　記憶装置の利用率

静的再配置で解決されていない問題の１つは次のことである．一旦ロードされるとプロセスのアドレス参照が記憶装置のある連続した領域に制限され，スワッピングに際しては，そのプロセスに物理プロセッサの制御が与えられるたびに，プロセスを主記憶の同じ領域にもどさなければならないということである．ベース・レジスタを使えば，この制約はもはやなくなる．プロセッサを主記憶内にないプロセスに切り換える場合，そのプロセスを入れる未使用の連続したブロックを主記憶内に見つけ出さなければならない．もしそのようなブロックが見つかれば，入ってくるプロセスを蓄えるために，補助記憶へ情報を退避させる必要はない．主記憶内にプロセスのための十分な未使用セルがあるにもかかわらず，セルが十分大きな連続したブロック単位で存在しない場合が良く生じる．この場合は，ベース・レジスタを用いるように設計されたシステムでは，次の３つのことを行なうことができる．(1)使用できる連続したブロックに適合できるプロセスを見つける．(2)十分大きな連続領域を作るために現在主記憶にあり，未使用領域に接しているプロセスの一部を補助記憶へスワップする．(3)主記憶において，使用中のブロックをまとめ，大きな未使用領域を作

る．（これを**コンパクティング**（compacting）という．）

図 2.4 (a) は，ある時間における記憶装置の様子を示している．記憶内に存する2つのプログラムと3つの**未使用領域**（hole）（空間）がある．記憶装置に3番目のプログラムCを持ってきたいが，Cはどの空間よりも大きく，空間1と2で得られる全領域よりも小さい．図 2.4 (b) はプログラムCに適合する十分な空間を作り出す1つの方法を示している．プログラムAを記憶装置の1番最初から開始するように全体を動かし，プログラムCのための十分な空間を作り出している．図 2.4 (c) は，プログラムCに適合する十分な空間を作り出す別の方法を示している．プログラムCの空間を作るために，空間1に隣接するプログラムAの一部を補助記憶に移したものである．

十分大きな連続領域を見つけるための一解決法は，多数のベース・レジスタを使って，存在する未使用空間にプロセスを細分してロードする方法である．しかしながら，この方法は特定部分の命令ごとに正しいベース・レジスタを参照しなければならないので，非実用的と思われる．プログラマまたはコンパイラは，いかにプロセスを分割し，どの部分にどのベース・レジスタを与えるかを決めなければならない．ロード時に命令をベース・レジスタ・アドレスに

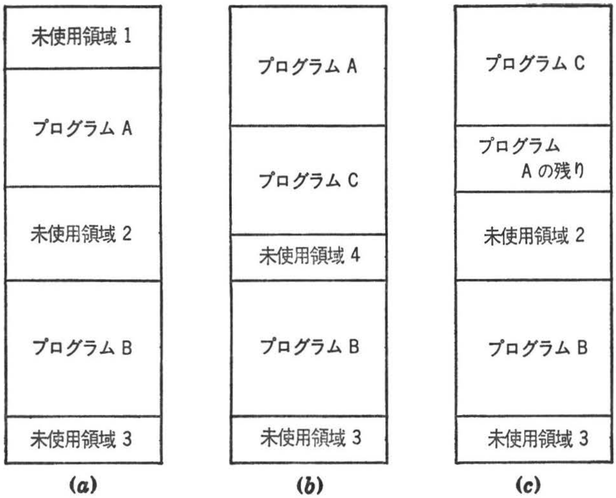


図 2.4　ベース・レジスタによる記憶割付け：(a) ある時点における典型的な記憶装置の様子；(b) コンパクティングによるプログラムCの記憶割付け；(c) プログラムAの部分的な除去によるプログラムCの記憶割付け．

割当てることは，そのプロセスを論理空間の一部に割当てることと同じ意味になる．解決しようとしている問題は，ロード時ではなく実行時に生じてくる．

このベース・レジスタの割当をシステムが動的に行なうことは容易ではない．なぜならば，その作業は非常に時間を浪費し，変更する命令を決めるのが複雑になるからである．プロセスが開始されると，システムはプログラムの特定部分が使っているベース・レジスタを知るために適当な番号を設定しなければならない．特定な部分を参照するすべての命令は，そのベース・レジスタ・アドレスを使わねばならない．また，共用されるプログラムは，それを呼出すプロセスあるいは同時に使われている別の共用されているプロセスのベース・レジスタと異なったものを使えるように約束をしておく必要がある．このような複雑なことをしても，システムが実行されだすと，プログラムの部分よりももっと小さな未使用空間がさらに増えるので，問題が解決されたとは言えない．このような記憶分割の問題をうまく避けるために，プロセスと物理記憶を一定の大きさの部分に分割することが行なわれている．これが2.4節で述べるページングの概念である．

記憶断片化の問題は計算のいろいろな場面で遭遇する．すなわち，記憶装置が動的に再配置されたり，もはや使っていない記憶領域を再割当する時は常に起こる．一般にこの問題は，記憶装置を一定の大きさのブロックの集まりとして割当てたり，リスト処理を用いたりして，与えられた要求に対し十分大きな未使用のブロックを作り出すために，記憶装置内で情報を移動することによって解決される．この重要な話題をさらに検討したい読者は，Knuth の名著[21]を参照されたい．

2.3.4　ま　と　め

ここでは，ベース・レジスタによってプログラムやデータを実行時に主記憶の任意の場所に動的に再配置できることを知った．2つのベース・レジスタを使えば，プログラムの最も単純な共用ができる．ベース・レジスタ方式の問題点は2つあることが指摘された．(1) ベース・レジスタを使っても，多数のプロセスによって必要な領域よりもさらに小さな未使用空間が動作中に生まれるために記憶装置を完全に利用できないので，コンパクティングやスワッピングが考え出された．これによって未使用領域を完全に使うことができるならば，この問題は理論的に解決できる．(2) 現在利用可能な機構では，論理アドレス空間の大きさは物理アドレス空間の大きさで制限される．オーバーレィ手法は

物理アドレス空間よりも大きな空間を実質的に作り出すものである.

前述の問題を解決するために，さらに複雑な動的再配置方式を導入し，その結果，付加ハードウェアやソフトウェアが複雑になり，費用がかかっても，実際にそれに応じてシステムの効率が向上し，応答性が改善され，プログラミングが容易になるかどうかは疑問である．この議論の賛否両論に関しては，さらに動的再配置の概念を概説した後で論じよう.

2.4 ページングによる動的再配置

2.4.1 概　　要

ベース・レジスタによる動的再配置はプログラムを主記憶の連続した領域に位置付ける必要があるが，これは小さすぎて十分使えない未使用領域を作り出すので，主記憶を完全に利用できない問題を起こす．しかしながら，プログラムや主記憶が小さなユニットに分割でき，そのプログラムのユニットを主記憶内の任意の場所の対応した大きさのブロックに入れることができれば，主記憶をより効果的に使える可能性が出てくる．**ページング**（paging）という名前は，記憶装置を一定の大きさに分割して実現する手法に対して与えられたものである．また，ページングの手法は物理記憶空間よりも大きな論理記憶空間を経済的に実現させる.

ページ化されたシステムは，物理記憶は一定の大きさの**ブロック**，通常 512，1024 または 2048 語に分けられるものと考える．**ページ**という用語は論理空間のユニットを指し，同じ大きさの物理空間のユニットを**ブロック**と呼ぶ．プログラムもまた物理記憶のブロックと同じ大きさのページに分割されると考えられる．したがって，このようなシステムにおけるアドレスは 2 つの数によって表現される．すなわち，(1)ページ・アドレスまたはページ番号，および(2)ページ内のライン・アドレスである．n ビットのアドレス・フィールドの計算機では，上位 p ビットがページ・アドレスであり，残り n−p ビットがライン・アドレスと考える．オペレーティング・システムは比較的大きなページをいくつか占める．新しいシステムでは，システムがページ・サイズを動的に変えることができる．使えるページの大きさが比較的小さければ（64，128，256語），システムは記憶装置をより完全に利用できる．ページの大きさが小さいと，記憶装置の利用がより効果的になる理由は次の通りである．一般に，どんなプロセスも 1 ページの整数倍の記憶空間をすべて必要としないので，プロセスの最

後のページは，そのために割当てられたブロックをすべて使っていない．平均してプロセスの最後のページは，与えられたブロックの半分が使われると考えるのが妥当である．ページの大きさが大きい程，空間の浪費は潜在的により大きくなる．読者は次の議論で分かるように，ページング機構には**ページ・テーブル**（page table）と呼ばれる1つのテーブル，あるいは，論理空間から物理空間へアドレス変換を行なうために，各ページごとに1つのエントリーをもったマップを必要とする．ページ・サイズが小さければ小さい程，与えられた論理アドレス空間に必要なテーブルは大きくなる．したがって，ページの大きさに関連した空間の浪費と，大きなページ・テーブルに蓄えたり，操作するために使うリソースとの間で，つり合いをとる問題が出てくる．使われないブロックによる空間の浪費の総計は，主記憶にはいると考えられるプロセスの数に依存する．

XDS-940 のページ・アドレス方式は，ページングの基本的な概念をよく示している．

2.4.2　XDS-940 のページング

XDS-940における1つのプロセスのアドレス空間は16K語†までの大きさをとり得る．XDS-940 の物理記憶は64K語まで拡張できるので，論理アドレス空間は物理アドレス空間よりも小さい．物理アドレス空間よりも大きな仮想記憶を作り出すより一般的なページ・システムについては2.4.4節で論じる．XDS-940 内にあるプロセスは2K語のページに分割され，記憶装置も同様に2K語のブロックに分けられる．940 における命令語のアドレス・フィールドは14ビットであり，3ビットのページ番号と11ビットのページ内ライン番号の2つの部分を含むものと考える．再配置機構（図 2.5）は，**記憶マップ**（memory map）と呼ばれる8個の6ビット・バイトを使っている．XDS-940 では，この記憶マップは2つの24ビット・レジスタで構成されている．それぞれのレジスタは，4つのマップ・バイトを持っている．これらのレジスタは，ページ番号を物理記憶のブロック番号に書換え（マッピング）するので，**実リラベリング・レジスタ**（real relabeling register）と呼ばれる．このマップ・バイトはハードウェアで0から7まで番号付けられ，論理ページに対応している．記憶アドレスに含まれるページ番号によって，1つの特定のマップ・バイトがアドレスされる．その特定のマップ・バイトの中は，論理ページのプログラムやデータを含む実際の物理ブロック番号である．たとえば，図 2.5

† Kの値は1024である．ゆえに，16Kは16×1024を意味する．

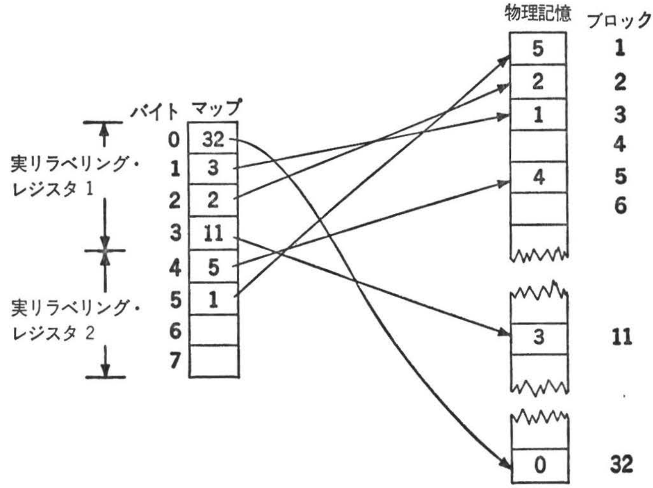


図 2.5 XDS-940 のページング

において論理ページ0は物理ブロック32の中にあり，論理ページ1は物理ブロック3の中にある．物理ブロック内に示した番号は，そのブロックが含む論理ページを示している．

論理アドレスは 図 2.6 に示すように，物理アドレスに変換される．

3ビットのページ番号が実際のページがはいっている物理ブロック番号をどのマップ・レジスタが持っているかを示している．マップ・レジスタは長さが

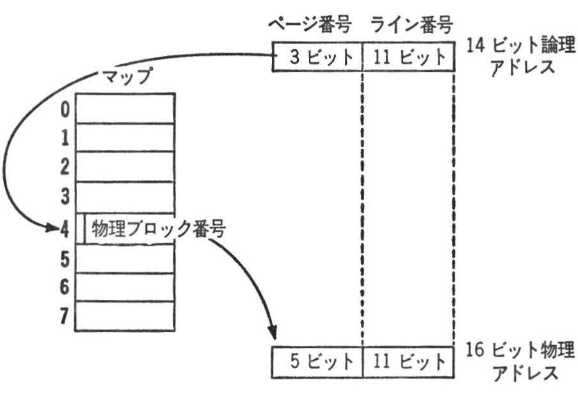


図 2.6 XDS-940 における論理アドレスから物理アドレスへのマッピング

6ビットで，それを図2.7に示す．5ビットが物理ブロック番号を持ち，1ビットは記憶保護（4.1.2節参照）のためのものである．物理アドレスは単に物理ブロック番号とライン番号を連結して作られ，その結果16ビット・アドレスとなる．16ビットで64Kの記憶装置をアドレスできる．

このハードウェア機構は大変簡単であるが，全体システムの一部として動作するためには，各プロセスの記憶空間を記録する（5.3.2節参照）付加的なソフトウェア・テーブルが必要になる．基本的な考え方は，プロセスが主記憶に持込まれる時，ソフトウェア・モニタは主記憶の状態を調べ，入ってくるプロセスの必要性を満たす未使用ページに関連して，必要なだけのページをスワップ・アウトする．その後，モニタは入ってきたプロセスの論理ページに使用可能な物理ブロックる割当て，そのページをそれらのブロックにスワッピングする．記憶マップを更新し，そのプロセスが以前実行された時に持っていた値をプロセッサ・レジスタおよびプログラム・カウンタに戻した後，そのプロセスを再開する．

2.4.3　記憶マップ

上記で概説した中で最も重要で一般的な概念は，**記憶マップ**（memory map）である．マップは，論理アドレス空間を物理アドレス空間に翻訳する．動的再配置の手法では，マップは記憶内のテーブルの集まり，またはハードウェア・レジスタの集まりである．静的再配置の手法では，マップはプログラムである．ベース・レジスタによる動的再配置の方法では，ベース・レジスタがマップである．ページングによる動的再配置法では，ページ・マップが多数のベース・レジスタを効率的に実現する一方法であると見なすことができる．ページングの処理はユーザやコンパイラには全く見えず，あたかも1つの連続した論理的領域を使っているかのように働く．記憶装置を均等に分割する機能は，主記憶をブロックに分けることによって可能になるが，あるブロックは一部しか詰まっていないかも知れないが，主記憶の全ブロックを使うことができることを意味している．それゆえ，ある時点でより多くのプロセス，または，プロセスの一部分を記憶装置に蓄えることができるので，スワッピングによる時間損

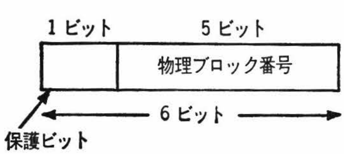


図2.7　XDS-940におけるマップ・バイト

失を減らすことが可能になる．論理アドレスを物理アドレスに動的に変換する記憶マップの概念は，計算機構成における1つの重要な改革である．

2.4.4 より一般的なページングの方法

XDS-940 ページング機構では，物理アドレス空間より大きな論理アドレス空間を簡単に実現することはできない．図2.8に，より一般的なページングの方法を示した．この方法では，マップは主記憶内の**ページ・テーブル** (page table) である．プロセスごとに1つのページ・テーブルが存在する．このページ・テーブルにおいて，インデックスとしてページ番号を使い，ページ検索することによって，特定なページに対応する物理ブロック番号を見つけ出す．各テーブル・エントリー（ページ・ディスクリプタ）における制御ビットはその中のエントリーによって表わされたページが主記憶内にあるか，あるいは補助

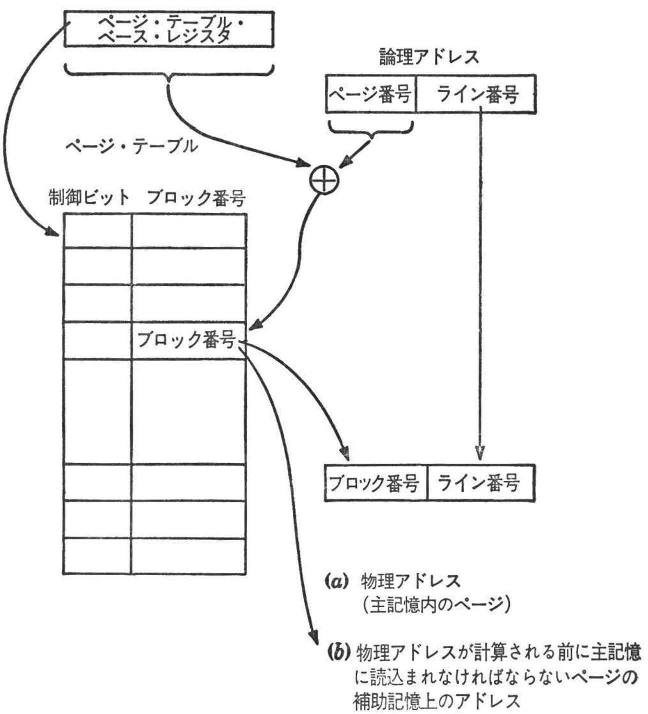


図 2.8 大きな論理記憶空間を許すより一般的なページング

記憶内にあるかを示すことに使われる．**ページ・テーブル・ベース・レジスタ**は，現在計算機の制御下にあるプロセスのページ・テーブルのベースを指している．論理アドレスからのページ番号はページ・テーブル・レジスタの内容に加えられるが，ページ・テーブル内のどの語がそのページが存在するブロック番号を持っているかを示す．そのテーブルのブロック番号の部分に入っている数は，主記憶内のそのページの実際の開始アドレス，またはそのページを見つけることができる補助記憶上の記憶場所を示す．もし制御ビットが後者の場合を示していれば，しばしば**ページ・フォルト**（page fault）と呼ばれるシステムへの呼出しが発生し，計算を再開する前にそのページを記憶装置にもってくる．この方法によれば，論理アドレス空間は物理アドレス空間より大きくできる．仮想記憶の大きさを制限する主な要因は，(1)作り出し得るアドレス・フィールドのビット数，(2)ページ・テーブルの大きさに対する経済的制約である．

いま述べた方法の難点は，すべての命令参照において，ページ・テーブルからブロック番号を得るために余分なアクセス時間を必要とすることである．1つの解決法は XDS-940 で行なわれたように，現在実行中のプロセスが使っているページ・テーブルをハードウェアで実現することである．しかしながら，大きな仮想記憶に必要なページ・テーブル（マップ）を直接ハードウェアで実現することはおそらく非常に高価になる．また，プロセッサがあるプロセスから他のプロセスに切換えるたびに，このハードウェア・テーブル（マップ）を更新するために多くの時間を要する．

余分の記憶参照を多くの時間にわたって除くために，**連想検索**（associative lookup）を用いた特別のハードウェア・マップを設けることがある[5]（図 2.9）．このマップは歴然としたアドレスではなく，連想，すなわち内容によってアドレスされるので**連想マップ**（associative map）と呼ばれる．この連想マップを使えば，論理アドレスのページ番号はマップ内のすべてのページ番号に対して同時に比較される．ページ番号が見つかれば，そのブロック番号が出力され，ブロック番号にライン番号を連結して，物理アドレスを作り出す．論理アドレスのページ番号とマップのページ番号の間で一致がとれなければ，主記憶内のページ・テーブルを参照しなければならない．その新しいページ番号とブロック番号の組はマップの中に入れられるが，その場合，マップ内のエントリーの1つが交換される．連想マップの大きさ，およびマップ内に既に存在するエントリーとの交換の方法は，このようなページング・システムの設計上の問題になる．このような問題を研究するには，ハードウェアの価格，およびプロ

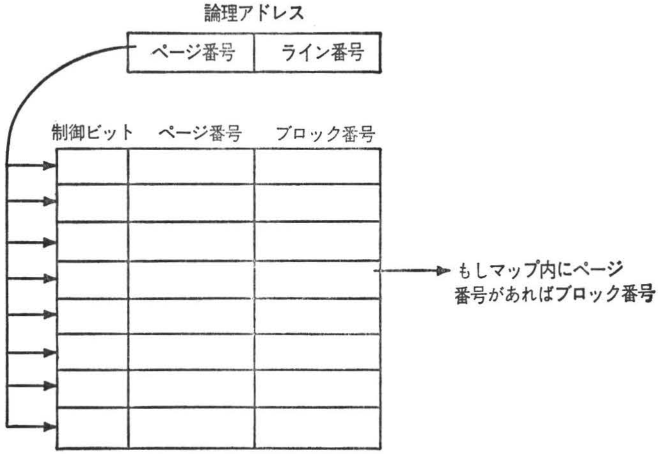


図 2.9　連想マップ

セスのページ参照特性のような要因を考慮せねばならない．関連した問題に対してConti[22] 等が検討したように，別の設計を研究するためにはシミュレーション手法が有用と思われる．

上述のようなページング方式を用いると，計算が始まった時点に全体のプロセスを主記憶にロードしておく必要はない．最初に必要なページだけをロードしておけば良く，主記憶内にないページに参照が起こると，管理ルーチンに呼出しを発生してページ・テーブルはその事実を示し，その後，管理ルーチンはそのページを主記憶に持込む．この**デマンド・ページング** (demand-paging) 方式の賛否両論については，5.3.3 節で論じる．

2.4.5　プログラムおよびデータの共用

先に述べたように，タイムシェアリング・システムではプログラムおよびデータを共用することが望まれる．共用はいろいろな方法で行なわれる．1つの方法は各ジョブが共用するプログラムやデータ構造を別々の写しで持つことである．別々の写しを持つことは，記憶装置への要求が増大し，スワッピング活動を増加させる結果となる．それゆえ，頻繁に用いられるプログラムやデータは，記憶装置内にたった1つ存在する状態で共用することが望ましい．図2.10は，ページ化されたシステムにおいてこのような共用をいかに行なうかを示している．図 2.10 (a)において2つのソース・プログラムAとBは1つのコンパ

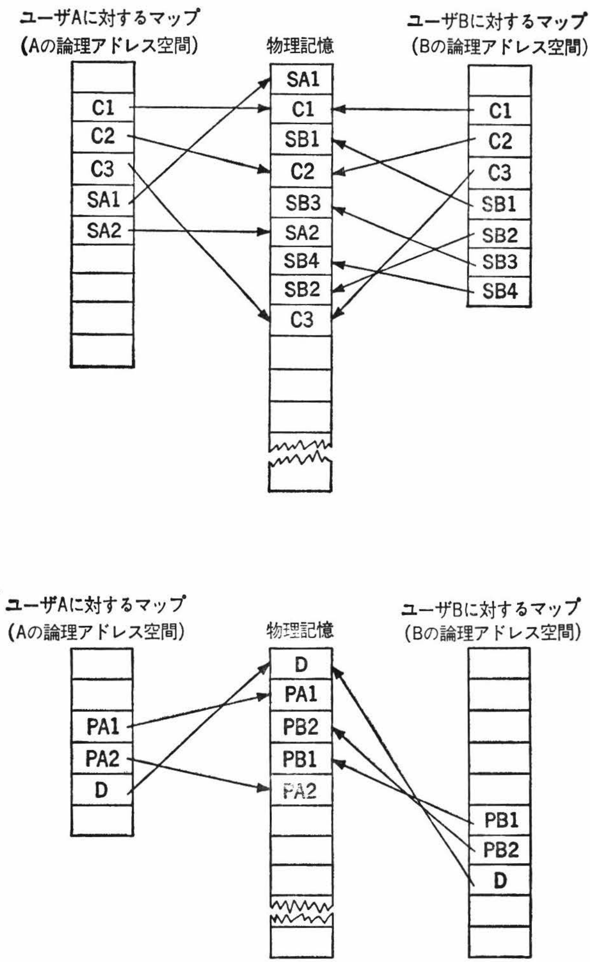


図 2.10　(a) ページ化された記憶装置内のコンパイラの共用；(b) ページ化された記憶装置内でのデータの共用.

イラCを共用している．コンパイラのページは C_1, C_2, C_3 と名付けられている．ここで注意すべきことは，物理記憶にはコンパイラの写しがたった1つしかないにもかかわらず，それを使う各ジョブのマップにはコンパイラのページのエントリーが入っていなければならないことである．また，さらに，各マップにおいてコンパイラのエントリーは同じ相対位置になければならない．同

様にソース・プログラムSAおよびSBも，その2つのマップの同じ相対位置から始まる連続的なエントリーを持たなければならない．このように連続エントリーを用いるのは，コンパイラがシステムに最初にロードされた時に，その記憶参照が固定領域の物理アドレス空間に割当てられるためである．この論理空間への割当てが行なわれる理由は，アドレス・フィールドがページ番号とページ内ライン番号を持ち，それぞれがロード時に値を与えられるからである．ページ番号はマップ内のエントリーに対するインデックスであり，そのエントリーはそのページが入っている記憶ブロックのベース記憶場所を含んでいる．コンパイラは実行されるたびに，マップ内の同じ相対位置を参照する．それゆえ，コンパイラを使うプロセスはすべてそれぞれのマップ内にコンパイラのためのエントリーを持ち，しかも，コンパイラがシステムにロードされた時に決められた位置に正しく入れておかなければならない．最初のロード操作では，コンパイラは論理空間のどこに入れても良い．静的再配置の場合の物理空間の位置付け，およびベース・レジスタやページングによるシステムの場合の論理空間の位置付を行なう**割当て処理**（binding process）としてのロード操作の概念は非常に重要で，十分理解して先に進むべきである．

各ユーザの論理空間において，同じ相対位置にコンパイラのページを置くかわりに，別のマップを持つ方法がある．コンパイラが呼び出される時に，このマップも呼び出される．コンパイラが共用される場合は，コンパイラのマップにユーザのソース・コードの論理的配置をいかに扱うかという難しい問題が起こる．コンパイルされる各ソース・コード・プログラムは，コンパイラの論理空間の同一場所から格納されなければならないので，コンパイラのマップは異なったジョブに制御が移されるたびに，変更されなければならない．もう1つの方法は，ある記憶参照はコンパイラのマップを通して行ない，その他の参照（ソース・コードを参照するもの）は，ユーザのマップを介して行なうように指示する方法である．これら2つの別法は複雑さを増し，しかも有用な利点をもたらさないように思われる．

さてここで図 2.10(b) に示すように，2つのプロセス P_A, P_B がデータを共用する場合を考えてみよう．データを共用するために，データを表わす論理ページDが各マップの異なった位置に入っている．データ自身がアドレスを含まない限り，この状態で良い．たとえば，Dを介して P_A と P_B にもどる間接アドレスが発生した場合は，P_A と P_B は論理空間の同じ相対位置になければならない．この記憶場所は，そのデータが作られた時に決定される．Dを介してD自身に間接参照を行なう場合，P_A および P_B は異なった相対位置を占めて

も良いが，Dは各マップの同じ相対位置になければならない．

2.4.6　まとめと結論

ここでは，ページングがどのようにして記憶装置を一様に分割し，記憶装置の利用をより効果的にする可能性を与えるかを知った．記憶マップに関する重要な概念を紹介した．記憶マップはハードウェア，ソフトウェアまたはその両方で実現される．

ある約束に従った時に，どのようにすればプログラムとデータが共用できるかを示した．約束とは何であるかというと，(1) 共用データ構造およびプログラムを使う各プロセスのページ・テーブルに，余分の空間を持たなければならないこと，(2) いかなるプロセスも，論理空間の同じ場所を占める共用ルーチンまたはデータ構造を2つ以上同時に使うことはないことである．論理空間が十分大きく，しかも同時に使われるすべての共用プログラムとそのデータを論理空間の異なった位置にロードするように十分注意すれば，2番目の問題が避けられる．この解には大きなページ・テーブル(マップ)が必要になる．2.4.4節で述べたページ・テーブルは，対応するブロック・アドレスを含むエントリーのインデックスとしてページ番号を用いた．この方法は非常に実現が簡単であるが，論理空間の各ページに対して1語のページ・エントリーが必要である．もしテーブルを擬似的な連想記憶，ハッシュ・テーブル (5.3.2 節参照) で構成し，ページ番号でアドレスすれば，論理ページ当たりたった1つのテーブル・エントリーで良くなる．この問題に関連する他の方法は，上に述べたページングに行く前に，アドレスを形成する過程で，ハードウェア・ベース・レジスタを使うことである．しかしながら，この方法は多数のベース・レジスタを必要とし，非常に注意深い管理とベース・レジスタの使用に関しておそらく実施不可能な約束を設定しなければならないので，非実用的であろう．

プログラムおよびデータの共用に関して完全な一般性を欠くほかに，ページングの方法は，リスト，記号テーブル，およびプッシュダウン・スタックのような実行中に大きくなったり縮んだりするデータ・ベースに対して，記憶装置の割付けをプログラマに行なわせる際に何の手助けにもならない．プログラマはこのような場合，マップが適当にセットされるように明確に計画をたてなければならない．XDS-940 システムでは，プロセスがシステムと交信してそのマップを変えられるようなシステム呼出しによって，ある程度この問題を解決している．このようにプロセスは，その論理アドレス空間の内容を制御できる大きな力を持つ．しかしながら，この機能はページングで得られる論理空間

の構造を基本的に変える場合よりも，オーバレイを容易に行なう場合に，より有用である．

ページングで得られる論理空間の構造は，ベース・レジスタで得られるものと同一である．いずれの場合もその構造は，動的に再配置可能な連続した大きなアレーである．ユーザにとっては，どちらの方法を使ってプログラムしても差異はない．ベース・レジスタよりもページングを使って得られるものは，記憶装置を分割することにより，記憶装置の利用率をより効果的にすることである．また，大きな仮想記憶を実現する実用的な方法でもある．さらに，ベース・レジスタの利用法に関する約束を設定する必要がないので，プログラムやデータの共用を設計する際に，より大きな融通性がシステム・プログラマに与えられる．

ベース・レジスタを使うよりも明らかにページングが不利になることは，(1)ページングではハードウェア記憶マップを効率的にする必要があること，(2)ページングの利点を完全にいかそうとすれば，ソフトウェアの実現が非常に複雑になること，(3)常駐ソフトウェア・システムが比較的大きく，主記憶をよけいに消費することである．ハードウェアの価格が低下している傾向があるので，ハードウェア・マップの余分の価格は全システムに比べて非常に小さくなってきている．ページングに必要なソフトウェアの複雑さと，システムの記憶領域は大きくない(5.3.2 節参照)．記憶装置をうまく使えば，より多くのプロセスを主記憶に同時に常駐させることができ，あるプロセスのスワッピング中に他のプロセスの計算を実行させる可能性を増す．しかしながら，高速の補助記憶機器を使うと，この有利性は減る． CPU や記憶の割付けおよびスワッピング・アルゴリズムを実験するシステム・プログラマに別の融通性を提供することを考えれば，ページング・ハードウェアに若干の費用をかけても割に合うように思われる．

2.5 セグメンテーションによる動的再配置

2.5.1 セグメンテーションの動機

次に述べる理由でセグメンテーションの概念はあまり広く理解されていないように思われる：

1. セグメンテーションに対する開発の動機が完全に明らかでない．

2. 計算機ごとに異なった形の概念が存在する．たとえば，IBM 360/67 システムにみられるセグメンテーションは，GE-645システムにみられるものに比べ，重要な差異がある．
3. 論理アドレス空間と物理アドレス空間の区別が必ずしも理解されていない．
4. ページング・システムおよびセグメンテーション・システムを実現する機構の多くは同じ様なものなので，2つの明確に異なった概念を互いにあいまいにする傾向がある．

本節では，最も一般的なセグメンテーションの形式を理解することによって，セグメンテーションに明確な解釈を与え，その概念の種々の方式の差を示す．また，論理および物理アドレス空間の差を再び力説し，ページングとセグメンテーションの区別を明らかにする．そして，セグメンテーションを開発するに至った動機を振り返ってみる．第6章においてファイル・システムを検討する際にはセグメンテーションを若干異なった観点から見ている．

今までの動的再配置の議論では，ベース・レジスタを使ったシステムにおいては**物理記憶**割付けの問題のあるものは避けられないことが分かっている．そこでページングを導入することによってこの物理記憶割付けの問題を解こうとしている．ところが，ページ化したシステムでは**論理記憶**割付けの問題が起こる．プログラムやデータを完全な一般性を失わずに共用したり，プログラマがあえて割付けの計画をしないでデータ構造を自由に伸び縮みさせることを許す場合にその問題が起こる．セグメンテーションはこれらの問題に対する解になる．

これらの問題は，非常に大きな論理空間を作ることができれば，ページ化したシステムでも解決できる．この場合，すべての共用プログラムはその空間内に決まった場所を占有し，データ構造は相互およびプログラムから十分離れたところに置き，任意に拡大できるようにしておく．ページ化したシステムでは，十分大きなアドレス・ビットと論理記憶から物理記憶にマップする大きなページ・テーブルを各ジョブごとに持たせることによって，この問題を解決することができる．このような解には，ある特殊な難しさをともなう．なぜならば，論理空間内で場所の取り合いを起こさないようにするために，各ジョブに対するマップを2.4.4節に述べたような形式では非常に大きくとらなければならないからである．このようにページ化したシステムでは，物理空間があたかも必要に応じて割付けられるように，論理空間を前もって割当てておかなけ

ればならないので，データ構造を任意に拡張できるようにしたい場合に論理空間に大きな隙を持たせることになる．また共用中のプログラムは論理空間において別の場所を広く占有するので，さらに隙ができる．このような隙ができるのは同時に使われるであろうすべての共用プログラムの組合わせを，論理空間の別の領域にロードしなければならないからである．したがって，上述の要求を満たすためには，論理空間の別々の広い領域にロードするプログラムの組合わせを選択することになる．さらに同時に使われる共用プログラムが論理空間内に同じ場所を占めないようにするために注意深い管理が必要になる．ページ・テーブルを作って，実際または見掛け上の連想記憶としてアドレスするようにすれば隙が起こらないのでその大きさを小さくできる．効率的な処理を行なうために，このような大きな連想マップをハードウェアおよびソフトウェアで実現しようとするものもまた難しく，今後の研究に期待されている．

ここでまとめると，完全な一般性をもたせてたった1つのプログラムおよびデータを共用したり，データ構造の拡大を許すためにページングを使うと次の理由で困難が生じる：

1. 大きな論理空間に固有のページ番号を保証するために，多数のアドレス・ビットを必要とする．
2. インデックス形式のページ・テーブルを使うと，おそらくあまり詰まっていないであろうが，大きなマップを必要とする．（共用プログラムおよびデータを使う各プロセスのマップに，共用プログラムおよびデータの各ページに対するエントリーを重複しておかなければならないが，効率的な連想マップを使えば，この議論は消滅してしまう．）
3. 同時に使われるプログラム同志は，論理空間内の同じ場所を占めないようにし，アドレス参照をもつデータを適当な場所に位置付けるように，システムが注意深い管理をする必要がある．

セグメンテーションの技術は非常に大きな論理空間を簡単に管理するために開発されたものである[6,11]．上に述べた問題を解決するためには大きな論理空間が必要である．ここで次のような簡単な区別をしておくと便利である．セグメンテーションとは，**論理空間**の割付けを取扱う技術で，**物理空間**の割付けを扱う概念であるページングとは異なっていることに留意すべきである．セグメンテーションの概念を説明するのに物理記憶割付けのページングを使うのは，両者の概念の区別を混乱させるので許すべきでない．

2.5.2 セグメンテーションの概要

静的再配置による物理空間の割付けは，プログラムやデータがシステムにロードされるときにアドレス参照を物理空間内の位置に割当ててしまうので，問題があった．ロードが終わると，すべてのアドレスは**絶対的な物理**（absolute physical）記憶場所になってしまう．この制約はベース・レジスタやページ化したシステムで除くことができた．その場合，物理アドレス参照をベース・レジスタまたはブロック番号のどちらかに対して**相対的**になるような機構を導入し，ベース・レジスタやブロック番号の内容は実行時までセットする必要をなくしたものである．しかしながら，ロード時の命令は使用するベース・レジスタまたはマップ・エントリーに密接な関係ができる．言い換えれば，ロードが完了するとすべてのアドレスは，**絶対的な論理**（absolute logical）記憶場所になってしまう．

セグメンテーションが解こうとする問題は，論理記憶の空間内で相対アドレスを許すことに関してである．これは論理空間を連続領域のかたまりに分け，1つのかたまり内のすべてのアドレスは，そのかたまりの最初の番地に関して相対的にしたということを意味する．そこで，各かたまりに対するベースの記憶場所を示すベース・レジスタがハードウェアまたはソフトウェアで必要になる．かたまり相互間の参照は適当にベース・レジスタを参照して，参照するかたまり内の相対アドレスを得るようにしなければならない．これらのベース・レジスタを実行時に割当て得るようなうまい機構が開発されている．連続記憶領域のかたまりを通常，**セグメント**（segment）という．このようにセグメンテーションの基本的な考え方はきわめて簡単であるが，実行時にベース・レジスタの割当てを許すような機構は比較的複雑である．

セグメントは，名前を持ったデータ要素（一般には計算機語）の順序だった集合である[6]．セグメント内の特定のデータ要素は，記号セグメント名およびそのセグメント内の記号データ要素名，〈S〉/〔α〕，で参照される．〈S〉と書けば，Sという名の記号セグメントを示し，〔α〕と書けば，そのセグメント内のαと名付けた記号要素を表わしている．記号セグメント名〈S〉は最終的（実行時）には，ベース・レジスタの番号に翻訳され，そのセグメント内の記号データ要素名〔α〕は，そのセグメント内の相対番地に翻訳される．言い換えると，セグメントは1次元のアレーであり，セグメント名はこのアレーの論理空間におけるアドレス（そのベース・アドレス）に関係する．そのセグメント内の記号要素名は，図 2.11 に示すように，セグメントの最初の番地に相対的

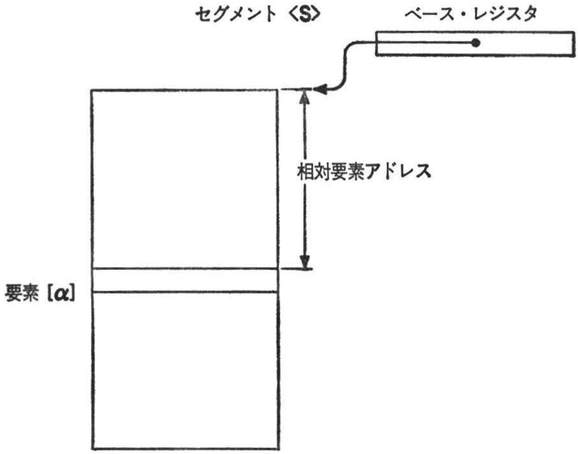


図 2.11 セグメント

に参照される要素のアドレスに関係する．

セグメンテーションは論理空間内の特定の要素を2つの名前で明白な形で参照するので，しばしば**2次元論理アドレス空間**と呼ばれる．ページング・システムはページ番号とライン番号の組のアドレスをもつが，それらはユーザには見えないので2次元とは言えない．一般に，ベース・レジスタおよびページ化されたシステムは1つのセグメントだけを許すセグメント化されたシステムと考えることができ，したがってそのセグメント名は明記しないセグメント化されたシステムと考えることができる．一般のセグメント化されたシステムでは，ユーザは2つの記述〈S〉/〔α〕を使ったアドレスでプログラムする．セグメントはプログラム，データ・アレー，記号テーブルまたはプッシュダウン・スタックのような関連した情報を自ら含んだ論理的な実体で，プログラマによって定義され，名前を付けられたものである．実際の実現においては，セグメントの長さに関しては上限があろうが，論理的には制約はない[7-10]．セグメントは必要に応じて伸びたり縮んだりすることができる．

セグメント化されたシステムはセグメントと呼ばれる異なったユニットの論理空間を提供するが，ベース・レジスタまたはページ化されたシステムは1つの同質ユニットの論理空間を提供する．各セグメントは，それぞれ別の論理的な実体である．セグメンテーションの概念を実現する際に解決すべき問題，ならびに決定すべきパラメータは次の通りである：

1. 記号アドレスの対〈S〉/〔α〕を物理的記憶場所にマッピングする方法

の開発
2. 許容すべきセグメントの数とその大きさの決定
3. セグメント同志をリンクする方法の開発
4. セグメントを共用する方法の開発
5. セグメントを保護する方法の開発

これらの問題に対する種々の解がハードウェアで実現されているし，文献にも紹介されている[7-16]．ここでは，現在用いられている機構の種類を良く表わしている Multics（GE-645）におけるセグメンテーション・システムの一実現法について若干詳細に述べ，他に提案されたり実現されたシステムに見られるセグメンテーションについて簡単に論じる．

2.5.3 Multics システムにおけるセグメンテーションの概要

次に述べる Multics システムの議論および説明は，Glaser およびその他[8]，Daley および Dennis[12]，ならびに Organick[15] による論文に基づいている．GE-645 の論理アドレス空間，すなわち仮想記憶は 2^{18} セグメントまでもつことができる．各セグメントの長さは，$2^{18}\times36$ ビット語まで許される．GE-645 CPU は，アキュムレータ，乗数/商レジスタを1つずつ，インデックス・レジスタ8個および1個のプログラム・カウンタを持ち，それぞれはいわゆる普通のレジスタの働きをする．

セグメンテーションの概念を実現するために付加レジスタがあり，それらは，**ディスクリプタ・ベース・レジスタ**（descriptor base register）および**プロシージァ・ベース・レジスタ**（procedure base register）を1つずつ，4つの**ベース・ペア・レジスタ**（base-pair register）である．ディスクリプタ・ベース・レジスタは，記憶装置内における後述する**セグメント・ディスクリプタ・テーブル**（segment-descriptor table）の記憶場所を示す．プロシージァ・ベース・レジスタは実行されるプログラムのセグメント番号（名）をもっている．ベース・ペア・レジスタはそれぞれセグメント名-セグメント内記憶場所の組を持ち，後に述べるように約束にしたがった特定の機能をはたす．最も一般的なアドレスの形式は，記号セグメント名-項目名の組である．最終的には，この記号名の組は物理記憶の1つの場所に変換されねばならない．これからの議論は，一般の記憶参照に対してこのマッピングがいかに行なわれるかを示している．

各プロセスに関連して1つのセグメント・ディスクリプタ・テーブル（しば

しばディスクリプタ・セグメントと呼ばれる)があるが，これ自身も1つのセグメントである．このテーブルは，それぞれの計算に対して**公開された**(known)セグメントごとに**セグメント・ディスクリプタ**(segment descriptor)と呼ばれる1語を持っている．それぞれのセグメント・ディスクリプタは，そのセグメントの物理空間におけるベース・アドレスを含んでいる．（セグメンテーションと一緒にいかにページングが用いられるかは，セグメンテーションの主要な概念が説明し終わるまでのばすことにする.）また，セグメント・ディスクリプタは記憶保護やここに直接関係のない他の目的のために使う制御ビットをも含んでいる．

セグメント・ディスクリプタ・テーブルは，図 2.12 に示すように，実際にはベース・レジスタのアレーである．ディスクリプタ・ベース・レジスタはセグメント・ディスクリプタ・テーブルを再配置可能にし，セグメント・ディスクリプタ・テーブルをシステム内で他のセグメントと同じように取り扱えるような一般的な機構を提供している．セグメント名〈S〉，あるいは後にわかるように実行時に得られ**セグメント番号**と呼ばれる〈S〉の変換は，セグメント・アドレス内の相対記憶場所と結合するセグメント・ディスクリプタ（ベース・レ

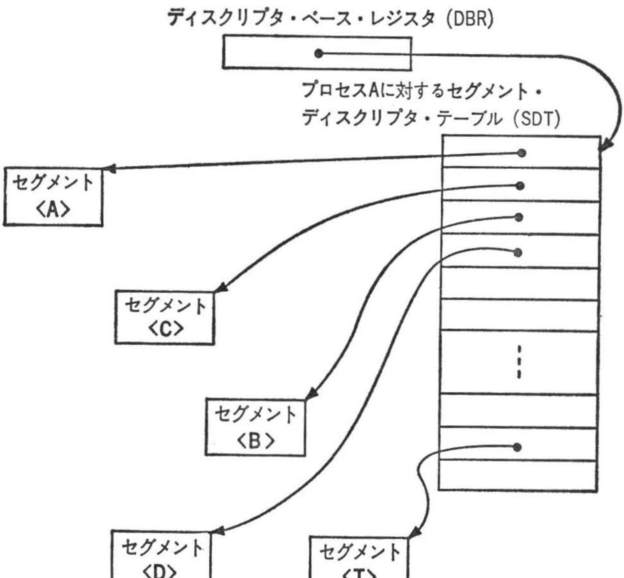


図 2.12 記憶装置内にあるセグメント化されたプロセス

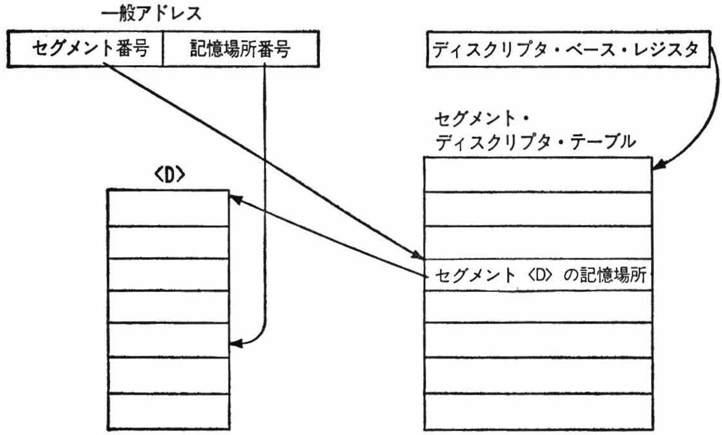


図 2.13　記憶場所の算出

ジスタ）を示し，その結果，図 2.13 に示すように物理空間のアドレスを作り出す．

セグメントがシステムにロードされる時は，すべてのアドレスはあるセグメントの最初の場所に対して相対的であるから，論理空間において制約がない．セグメント・ディスクリプタを作り出して，プロセスに対してセグメントが公開された時に初めて，そのセグメントの記憶場所と論理空間の結合が行なわれる．この結合は，セグメント名を新しく作り出したセグメント・ディスクリプタを示すセグメント・ディスクリプタ・テーブル内のインデックスに変換することによって行なわれる．同じセグメントを使ってもプロセスが異なっていれば（ある場合には同じプロセスでもあり得る），セグメント・ディスクリプタ・テーブル内の異なった場所にそのセグメント・ディスクリプタを持つ．この様子を図 2.14 に示す．セグメント・ディスクリプタの割当ての機構に関しては，リンク操作に関する節で述べる．

セグメント・ディスクリプタ・テーブル内の場所は，論理空間内の場所と考えることができる．異なったプロセスに対するディスクリプタ・テーブル内の異なった場所に同一セグメントが1つのディスクリプタを持つことができるが，それは実行時に割当てられる．また，ある場合には，同一プロセスに対するディスクリプタ・テーブルの異なった場所にそれを持つことができる．それゆえ，プログラムやデータが最初に参照された時に，プロセスの論理空間内の

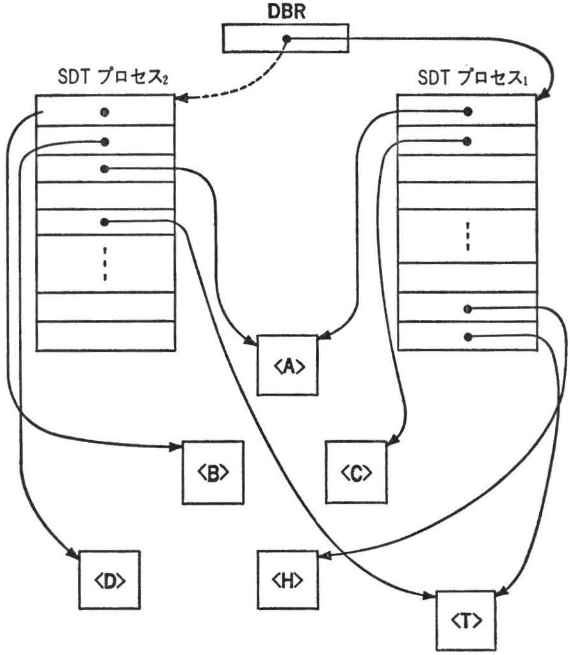


図 2.14　記憶共用セグメント内にある2つのプロセス

場所にそれらを割当てることができるようなシステムにすることができる．この場合，そのプログラムやデータが他のプロセスの論理空間内に占める記憶場所とは無関係である．ページ化したシステムあるいはベース・レジスタ・システムでは，たった1つの共用情報を用いるプロセスの全アドレス空間において同一の論理空間の記憶場所が割当てられるが，それはその情報がシステムにロードされる時である．リンケージ・セクション（2.5.5 節）における記憶場所はセグメント番号を保持しているが，セグメント・ディスクリプタ・テーブル内の場所（論理空間内の場所）を示しているので，論理的なベース・レジスタと考えることができる．セグメント・ディスクリプタは物理空間内のベース・アドレスを持っている．言い換えれば，セグメント・ディスクリプタは実行時までプロセスのセグメントに割当てられないベース・レジスタであり，セグメント名は実行時に1つのセグメント番号，またはそのセグメントを使う各プロセスで一般に異なるセグメント番号に割当てられる．

セグメンテーション・システムとして働かすためにはセグメント同志が適当にリンクできなければならない．Multics においては，このリンクは完全な一

般性を失うことなく，セグメントの共用を実現できるように工夫されている．(2.5.5 章参照)

2.5.4　Multics システムのアドレス方式

ここで Multics システムにおけるアドレス方式について若干詳細に解説しよう．ソフトウェアおよびハードウェアは相互に密接な関係があるので，両者の概念について述べる．まず最初に，プロセスが要求するセグメントはすべて公開されており，セグメント・ディスクリプタ・テーブルにそのエントリーがあるものと仮定する．セグメントを公開するステップについては後に述べる．記号名〈S〉を持つセグメントに対するディスクリプタのセグメント・テーブル内における相対的な記憶場所は，セグメント〈S〉の**セグメント番号S#** と呼ばれる．セグメント〈S〉は，いくつかのプロセスによって共用される可能性があるが，それぞれのプロセスで異なったセグメント番号を持っている．すなわち，セグメント〈S〉のディスクリプタは，各計算に対するセグメント・ディスクリプタ内の異なった相対的な記憶場所にある．図 2.12 は記憶装置内にセグメント〈A〉，〈B〉，〈C〉，〈D〉および〈T〉をもつプロセスがあることを示している．ここでディスクリプタ・ベース・レジスタはセグメント・ディスクリプタ・テーブルのベースを指していることに注意してほしい．第2.14図は，セグメント〈A〉および〈T〉を共用し記憶装置内に常駐する 2 つのプロセスを示している．各プロセスは，〈A〉および〈T〉に対して，各セグメント・ディスクリプタのセグメント・ディスクリプタ・テーブル内に，異なった記憶場所を持っている．各プロセスは，セグメント〈A〉，〈T〉を異なったセグメント番号で参照する．ディスクリプタ・ベース・レジスタの内容は異なったプロセスに計算機の制御が移るたびに変更される．それゆえ，その内容は現在実行中のプロセスに対するセグメント・ディスクリプタ・テーブルの最初の場所を指示する．

記憶装置を参照するには，2 つの番号が必要である．すなわち，(1)セグメント・ディスクリプタ・テーブル内の所望のセグメント・ディスクリプタを参照するためのセグメント番号，および(2)そのセグメント内の記憶場所番号である．セグメント内のセグメント番号 - 記憶場所番号の組を**一般アドレス** (generalized address) と呼ぶ．S#/α という書き方が使われるが，S# はセグメント番号であり，α は記号要素名〔α〕の〈S〉における記憶場所である．ここで，一般アドレスが形成される方法について述べよう．まず最初に命令取出しを考えてみよう．実行するセグメントのセグメント番号が，**プロシージァ・ベース・レジスタ**と呼ばれるレジスタの中に入っている．セグメント内の記憶場所は

プログラム・カウンタに入っている．プログラム・セグメントが，そのセグメント内にある命令のシークエンスを実行している間は，プロシージァ・ベース・レジスタの内容は変わらない．

GE-645 の命令では，**外部フラグ**（external flag）と呼ばれる1ビットがあるが，命令が実行中のセグメント内にオペランド参照する場合には**オフ**になっている．また，実行中のセグメントの外のセグメントを参照する場合は**オン**になっている．外部フラグが**オフ**の場合は，そのセグメント番号はプロシージァ・ベース・レジスタから得られ，セグメント内の記憶場所は実行中のセグメント内のインデックス・アドレス方式および間接アドレス方式を含む通常のアドレス変更過程を経て得られる．

外部フラグが**オン**であるような命令は図 2.15 に示すように一般アドレスを

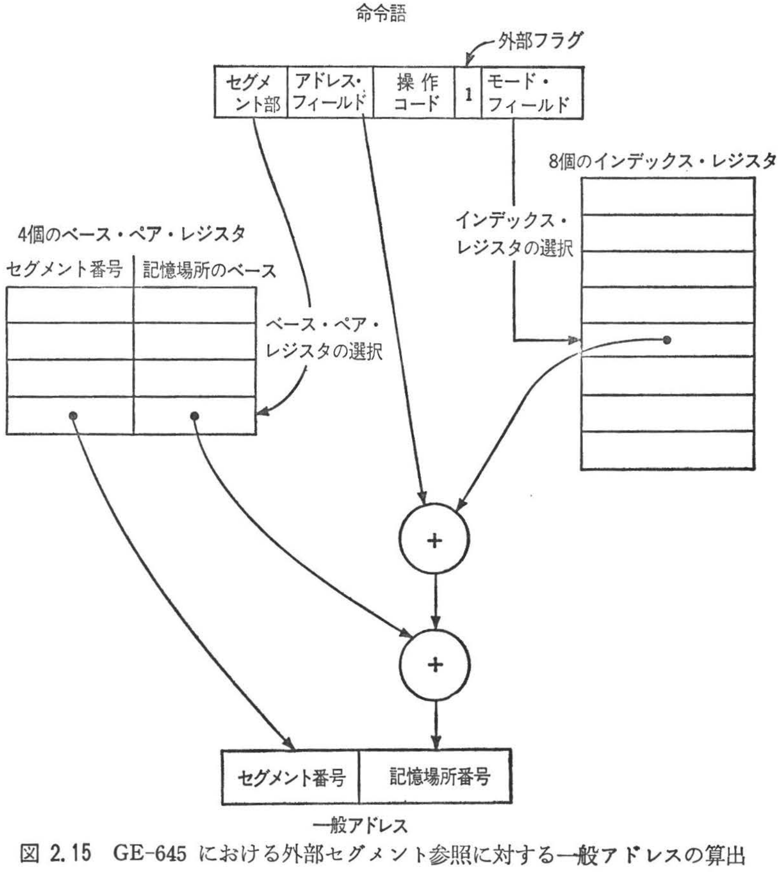


図 2.15 GE-645 における外部セグメント参照に対する一般アドレスの算出

作り出す．4つのベース・ペア・レジスタのうちの1つを指すためにセグメント・タグが用いられる．そのレジスタの一部には，セグメント番号が含まれ（いかにしてそうするかは後述する），他の部分にそのセグメント内の記憶場所のベースが入っている．そのベースとおそらくインデックス変更されるであろうが，そのアドレス・フィールドを組合わせて，セグメント内の記憶場所を作り出す．図 2.13 は，一般アドレスから物理空間内の実際のオペランド・アドレスをいかに得るかを示している．セグメント番号とディスクリプタ・ベース・レジスタ内の番号が加えられ，セグメント・ディスクリプタ・テーブル内の所望のセグメント・ディスクリプタを指す．そのディスクリプタはセグメントのベースを指している．セグメント・ディスクリプタ内のアドレスに記憶場所番号が加えられ必要な記憶場所を得る．

この機構は，データ構造を伸縮する問題の解になるが，プログラムは物理記憶空間の割付けに関しては心配する必要はない．データ構造がセグメントであれば，その構造は現実に許され得る最大セグメントの大きさまで伸ばすことができるし，また自由に縮めることもできる．そのデータ構造にその時要求する分だけの物理記憶空間が用いられるだけの話である．実現には，そのセグメントはページ化され，セグメント・ディスクリプタ内のポインタはセグメントのページに対する**ページ・ディスクリプタ**が入っている**ページ・テーブル**のベースを示す．ページ・ディスクリプタは主記憶または補助記憶内のアドレスを持ち，そこに所望のページが格納されている．記憶場所番号はページ番号とそのページ内のライン番号の2つの部分に分けられる．実際の物理アドレスは図 2.16 に示すように，テーブル参照を2回行なって得られる．

上述のままの方式であると，それぞれのアドレスを計算するのに時間がかかって望ましくない．この問題を避けるために，記憶マップとして小さな連想記憶をプロセッサ内に組込む[8]．連想記憶については 2.4.4 節のページ化した計算機で述べたが，連想記憶のキーはページ番号であった．ここでも概念は同じであるが，比較的長いキーが必要である．すなわち，ページおよびセグメント番号の組合わせである．上に述べた2回のテーブル参照が行なわれると，直ちにそのページおよびセグメント番号が実際のページ・アドレスとともに連想記憶に書き込まれる．したがって，このページおよびセグメント番号が再び使われると，そのページ・アドレスは連想記憶から直接読み出され，セグメント・ディスクリプタ・テーブルおよびページ・ディスクリプタ・テーブルの参照は省略される．

連想記憶は大きさに制限があるので，連想記憶が一ぱいに詰まり，そこにないページがアドレスされた場合に，どのページ-セグメント番号と置き換える

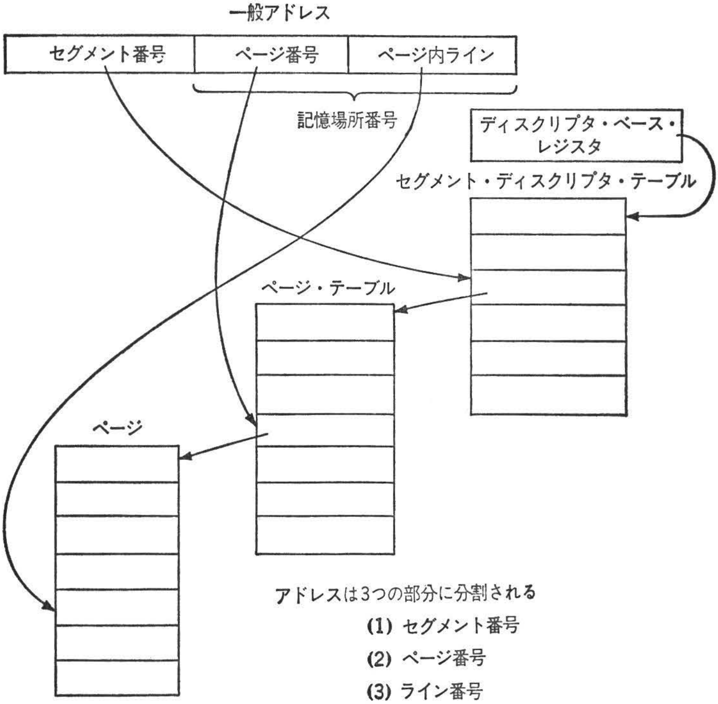


図 2.16　ページングによる一般アドレスから物理記憶場所の算出

かを決めるハードウェア・アルゴリズムを盛り込まなければならない[7,8].

ここで間接アドレス方式にもどってみよう．間接アドレス方式は，以下に述べるセグメントのリンク方式に広く用いられるので，アドレスを一般化するという面からも間接アドレスの方法は有用である．GE-645で間接アドレスが指定されると，2語組合わせて読出してくる．その記憶場所は，アドレス計算によって得られる（図 2.15 および図 2.16 で図示し，上に述べた通りである）．この2語は最初の語のあるビットのセットの状態に応じて，次の3通りに解釈される：

1. 間接語として同じセグメントからオペランドを読出し，最初の語が記憶場所を与える．
2. 最初の語に書かれているセグメント番号のセグメントからオペランドを読出す．2番目の語がそのセグメント内の記憶場所を与える．

3. 最初の語が示すベース・ペア・レジスタ内のセグメント番号によって指示されたセグメントからオペランドを読出す．2番目の語がそのセグメント内の記憶場所を示す．

間接アドレスおよびその処理は何レベルでも良い．したがって，複数個のセグメントを通るオペランドに至る前に1つのセグメント内で，いくつかのレベルの間接アドレスを行なうことがある．また他のセグメントにいく前にもいくつかのレベルをすませることもある．実際にGE-645の間接アドレス方式は概念的には普通の計算機と同じであるが，各レベルで記憶場所番号の他にセグメント番号を得るところが異なっている．

2.5.5 Multics システムにおけるセグメント・リンクの概要

上に概説したGE-645のアドレス方式で，セグメントをリンクする方法について述べよう．静的再配置またはページングやベース・レジスタによる動的再配置機能をもつシステムでは，プログラムやデータはすべて物理空間または論理空間にロードされる時，相互にリンクされる．この方法からは，ロード時に物理空間または論理空間に対し，それぞれ制約を生み出す．静的再配置では，実行時に物理空間の割付に関して問題を起こした．ベース・レジスタまたはページ化したシステムでは，論理空間の割付けに関して問題を起こした．Multics のセグメンテーション機能は，実行時にセグメント・ディスクリプタ（ベース・レジスタ）を割当てることによって，論理空間の割付を可能にすることを既に述べた．このセグメント・ディスクリプタおよびそれに対応するセグメント番号は，プロセスがセグメントを要求した時にだけ割当てられるべきものである．この時点以前では，所望のセグメントは補助記憶のどこかに格納されているファイルにすぎない．以下の議論は，Daley および Dennis による有名な論文[12] に基づいているが，セグメント・ディスクリプタを割付け，プロセスのセグメント間をリンクするために Multics で用いられた機能を概説している．

Multics のセグメンテーション機構の目標の1つは，1つのセグメントがそれぞれのプロセスの論理空間内の異なった場所に存在しうるということである．このことは，あるセグメントを共用するプロセスは，それぞれ異なったセグメント番号でそれを参照できることを意味している．プログラマは実行時に割当てられるセグメント番号を知らないから，記号セグメント名でセグメント間の参照がすべてできなければならない．したがって，リンケージ機構は記号セグメント名をセグメント番号に変換できなければならない．

リンケージ機構に望まれるもう1つの機能はセグメント内の項目に対する参照を相対記憶場所よりもむしろ記号名で行なえるようにすることである．このような機能があれば，セグメントを変更し，また再びコンパイルすることもでき，しかもその名前や項目名を変更しない場合は，それを参照する別のセグメントでは何の変更も生じない．プログラム・セグメントを共用しようとすると，リンケージ機構に種々の制約がでてくる．プログラムを共用可能にするためにはリエントラントでなければならない．これはプログラムの部分は実行中に変更することができず，共用プログラム・セグメントを呼出す各プロセスには，それぞれ個有のデータ領域を一時記憶として備えなければならないことを意味している．共用プログラムは，もとの呼出しているプログラム・セグメントが知らない他のプログラム・セグメントを呼出すことがあるので，一時記憶の割当ては前もってしなくても良いように取り扱えなければならない．これらの要求等に関してはさらに後で論じる．

リンケージ処理が始まる前に記号名で参照されたセグメントは，それを参照するプロセスのセグメント・ディスクリプタ・テーブル内で，セグメント・ディスクリプタとして表現されねばならない．言い換えると，参照されるセグメントにはセグメント番号を割当てる必要がある．セグメントに対してセグメント・ディスクリプタを設定する操作をセグメントを**公開にする** (known) という．各プロセスには**公開セグメント・テーブル** (known-segment table) と呼ばれる記号テーブルが付随する．セグメントにセグメント番号が割当てられると，公開セグメント・テーブルに1つのエントリーが作られる．それは，記号セグメント名-セグメント番号の組から成っている．

記号セグメント名を持つ命令に初めて会うと，管理ルーチンの呼出しが起こり，命令語の記号セグメント名と同じものを公開セグメント・テーブルに捜す．それが見つかれば，管理ルーチンは対応するセグメント番号を得て，リンク処理にはいる．見つからなければ，セグメント・ディスクリプタを作り，セグメント番号を割当て，そして公開セグメント・テーブル内にエントリーを記入する．このセグメント・ディスクリプタは最初，セグメント・ディスクリプタ・テーブル内の未使用の記憶場所に入れられる．セグメント番号は，対応するディスクリプタに対するこのテーブルへのインデックスになる．セグメントのセグメント番号はプロセスごとに変わり，同じプロセスでも処理中に変わることがある．

作られたセグメント・ディスクリプタは，セグメントがどこにあるかを示していなければならない．公開セグメントの検索で，命令で与えたセグメント名

と同じものが見つからない場合は**ファイル・システム**と呼ばれる管理ルーチンが呼び出され，そのセグメントを捜す．Multics のファイル・システムについては，第6章でさらに述べる．ファイル・システムは，現在活動中のプロセスに対して公開される全セグメントの記憶場所をもつテーブル，ならびに必要であれば**ファイル・ディレクトリ**と呼ばれる他のテーブルを調べ，そのセグメントの記憶場所を見つけ出す．そしてセグメントの記憶場所が適当なセグメント・ディスクリプタに書き込まれる．記憶システムでの情報の移動ならびに記憶の割当てはリンク操作と異なった話題であり，ここではこれ以上論じない．

セグメントに対して一番最初の参照が行なわれた時に，何が行なわれるかを考えてみよう．たとえば，次のようなセグメント〈A〉で実行されるコードを考えてみる．

〈A〉

……………………………………………

load accumulator　〈S〉/〔α1〕
store accumulator　〈S〉/〔α2〕

……………………………………………

2次元記号アドレスは〈A〉と異なるセグメントを参照している．

ここまでのところで，記号アドレスを使う要求によっていくつかの問題が生じることがわかる．読者にも気付いていると思われる1つの問題は，記号アドレスはどこに格納されているかである．命令のアドレス・フィールドに記号-アドレスの組を入れる余地はなく，それぞれ1字ずつしか入らない．もう1つの問題は命令への最初の参照に遭遇した際に，GE-645のアドレス機構をどのように使うかである．セグメント番号が一旦与えられると，以後の参照をできるだけ効率化するために，ハードウェアの一般アドレス機構を使いたくなる．システムは記号アドレスをそのハードウェアが使える情報に変換しなければならない．プログラム・セグメント〈A〉は純手続（pure procedure）であることが望まれるので，その内容をリンケージ処理で変えることはできない．これは，外部参照に関連して変更する必要のある情報は，セグメント〈A〉の外に出すことを意味している．

Multics のリンケージ機構は，リンケージ・セクションと呼ばれる1組の情報をそれぞれのセグメントに付随させることによって，これらの問題を解決している．セグメント〈A〉に対する**リンケージ・セクション**（linkage section）は〈link A〉と呼ばれる．このリンケージ・セクションはアセンブラまたコンパイラで作り出される．コンパイラは，それぞれのセグメント外の参照に関連

してリンケージ・セクションに1対の間接語を作り出す．外部参照を行なう命令はこれらの語を参照する．記号参照が最初に起こった時に管理ルーチンに飛び越しできるように1対の間接語の中にあるコードがあり，それによって参照するセグメント・アドレスが記号形式の場合は管理ルーチンへ飛び越しを起こす．この場合，1対の間接語には記号列のポインタが実際に含まれている．

記号セグメント名に遭遇すると，管理ルーチンは公開セグメント・テーブルの中に命令にあるものと同じ記号セグメント名を捜す．名前が見つかると最初の間接語に関連するセグメント番号を入れる．名前が見つからないとセグメントを公開する操作にはいる．セグメント名における記号記憶場所，上の例では〈α1〉はセグメント内の相対記憶場所に変換されねばならない．各セグメントをそれ自体で可能な限り完全にするために，外部セグメントによって参照されるすべての記号に関する記号テーブルを持ち歩く．この記号テーブルは，システムにわかる標準的な記憶場所に置かれる．したがって，一旦セグメント番号が分かると，そのセグメントが位置付けられ，セグメント記号テーブルを捜して，たとえば，〈α1〉の記憶場所番号を見つける．次にこの番号は，リンケージ・セグメントで与えられた命令に関連する2番目の間接語の内容を置き換える．間接語のコード・ビットを変更して，今や一般アドレスがそこに存在し，2.5.4 節で間接アドレスについて述べた方法で，記憶参照が許されることを示すようにする．この命令に対する以後の参照は，リンケージ・セクションを通して間接アドレスによって自動的に外部アドレスを参照する．管理ルーチンは一番最初の参照にかかわりあうだけである．

それでもまだ問題がある．リンケージ・セグメントがどこにあるかをセグメントはいかに知るかが問題になる．プログラム・セグメントをリエントラントにするために，外部参照をする命令のアドレス・フィールドは固定したディスプレースメントしか含むことができない．固定したディスプレースメントは，その命令に関連したリンケージ情報が格納されているリンケージ・セクションの最初の場所に対する相対アドレスである．リンケージ・セクションのベースの記憶場所を示す別の一般アドレス・ポインタも必要である．リンケージ・セクションを示すために使う一般アドレスは，一般アドレスを作る際には，ハードウェアが用いるプロセッサ・レジスタの中に入っている．**リンク・ポインタ**（link pointer）と呼ばれるベース・レジスタの1つが，この仕事のために割当てられる．リンケージ・セクションは，それぞれ別のセグメントと考えることができるが，場合によっては，あるプロセスの複数個のリンケージ・セクションが1つのセグメントの中にあることもある．プログラム・セグメントがい

かにデータ・セグメントにリンクするかが分かったので，次にプログラム・セグメント同志のリンクがいかに行なわれるかを調べてみよう．

2.5.6 Multics におけるプログラム・セグメント間のリンク

プログラム・セグメント間をリンクするためには，前節で述べたいくつかの機構が必要である．あるプログラムが他のプログラムを呼ぶとき，引数の引き渡し，プロセッサの状態の退避と復帰，および制御の返還に関する約束が必要である．Multics のプログラムはすべてリエントラントであるので，呼ばれたプログラムもまた呼んでいるプログラムに関連する固有記憶領域をアクセスすることがある．この固有記憶領域は，呼んでいるプログラムの中に含めることはできない．なぜならば，それ自身がリエントラントであり，いくつかのプロセスがそれを使ってるかもしれないからである．それゆえ，一時記憶領域は呼び出すプログラム・セグメントの外になければならないし，そのプログラムを使う各プロセスに関連して備えなければならない．いかにしてこの一時記憶領域を割付け，プロセスのプログラムの中における実行過程の詳細をプログラマが知らなくても済むようにするのだろうか．呼ばれたプログラムが最後に呼び出しているプログラムの呼び出しのところにもどる場合，呼び出されたプログラム構造の中の実行過程は，**プッシュダウン・スタック**と呼ばれるデータ構造で表現することができる．Multics では，プログラム・セグメント間でのリンケージはプッシュダウン・スタックを使うことになっている．プッシュダウン・スタックはシステム・プログラミングでしばしば使われる重要なデータ構造である．ここでは，Multics におけるそれらの用法を概観するだけにとどめる．プッシュダウン・スタックの用法に関する背景と動機付けは，Knuth[21] および Wegner[23] による名著に載っている．

プッシュダウン・スタックは後入先出リストである．Multics ではスタックはセグメントで，各プロセスに対して1つのスタック・セグメントがある．そのスタック・セグメントの現在のエントリーを指すために，ベース・ペア・レジスタの1つが割当てられている．これを**スタック・ポインタ**と呼んでいる．スタック内のエントリーは，1ブロックの連続語からできている．このスタック・エントリーは，引数の引き渡し，制御の復帰，プロセッサの状態の退避，および一時記憶領域に用いられる．スタック・ポインタに相対的にスタック・エントリー内に情報が入れられたり取り出されたりする．引数は CALL と同じセグメント内に CALL に相対的に置き，引数リストを示すベース・ペア・レジスタを使って引き渡すことができる．図 2.17 はこの機構を簡単に示して

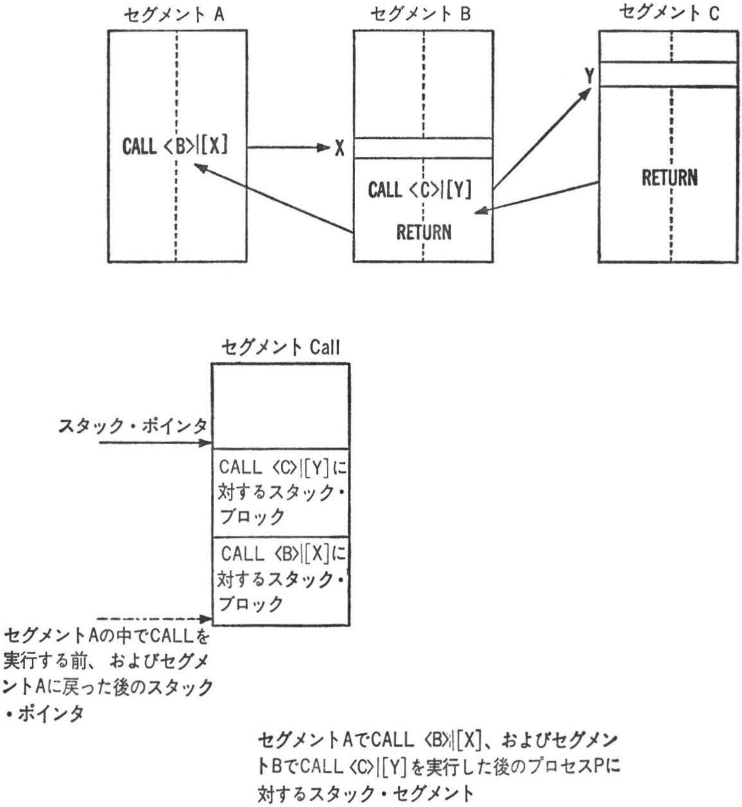


図 2.17 プッシュダウン・スタックによるプログラム・セグメントの CALL リンケージ

いる．セグメントA内の CALL 〈B〉/〈X〉 が実行される前は，スタック・ポインタは破線矢印で示すようにスタック・セグメントのエントリーのベースを指している．CALL が実行されると，スタック・ポインタは次のエントリーのベースを指すように変わる．セグメントBの CALL 〈C〉/〈Y〉 が実行された後は，スタック・ポインタは実線矢印で示したようになる．プログラム・セグメントはプロセッサの状態を格納し，引数を呼び出し，スタック・ポインタに対して相対的に割当てられたスタック・エントリー内の語からなる一時記憶領域を使う．

RETURN が実行されると，現在のスタック・エントリーに格納されている適当な情報からプロセッサの状態を復帰し，スタック・ポインタは自動的にそ

の前のエントリーを示すように修正される．図 2.17 では，2つの RETURN が実行された後のスタック・ポインタを破線矢印で示している．プログラムの CALL リンケージにプッシュダウン・スタックを使うのは極めて普通で，リエントラント・プログラムが簡単に書けるようになる．すべてのプログラムは完全に再帰性がある．すなわち，そのプログラム自身を呼び出せる．スタック・セグメント内の記憶領域のブロックは CALL および RETURN 機構によって自動的に割付けられたり，取り消されたりする．

もう1つの点を検討すべきであろう．直接であれ CALL を介してであれ，セグメント間の飛び越しが行なわれる時は，現在実行中のセグメントのリンケージ・セクションを示しているベース・ペア・レジスタは変更されなければならない．また CALL の場合は退避されなければならない．リンク・ポインタを変えるに必要なコードは，当然のことながら，これから入ろうとしているセグメントに固有のものである．Multics ではこのような場合を取り扱うために，2.5.5 節で述べたリンケージのやり方を次のようにプログラム・セグメント・リンケージとして変更している．いまセグメントAからセグメントBへの飛び越しが行なわれるものと仮定する．セグメントBへのリンケージはセグメントAのリンケージ・セクションを通して間接的に行なわれるが，リンク・ポインタは変わらない．セグメントBのエントリー・ポイントで飛び越しのためのコードを入れずにリンケージ・ベース・ペア・レジスタを自動的に変更させるには，セグメントAのリンケージ・セクションからセグメントBのリンケージ・セクションへリンケージが行なわれるようにする．リンク・ポインタを変えるためのコードがセグメントBのリンケージ・セグメントに入れられ，セグメントBのエントリーへ飛び越す．図 2.18 はこの機構を示している．

CALL が生じるとこの機構が用いられ，ベース・ペア・レジスタを含むプロセッサの状態をスタック・セグメントに退避するために必要な操作も付加的に行なわれる．

2.5.7 プログラムとデータの共用

さてここで，セグメンテーションによって，全く一般性をそこなわずに，プログラムおよびデータの共用をいかに可能にするかを調べてみよう．興味ある問題は，純手続である共用プログラムが現在制御下にあるプロセスの適当なデータ・セグメントをいかにアクセスするかである．この問題は，共用プログラムに付随してそれを用いる各プロセスのリンケージ・セグメントによって解決される．つまり別の見方をすれば，共用セグメントを使うプロセスはすべてそ

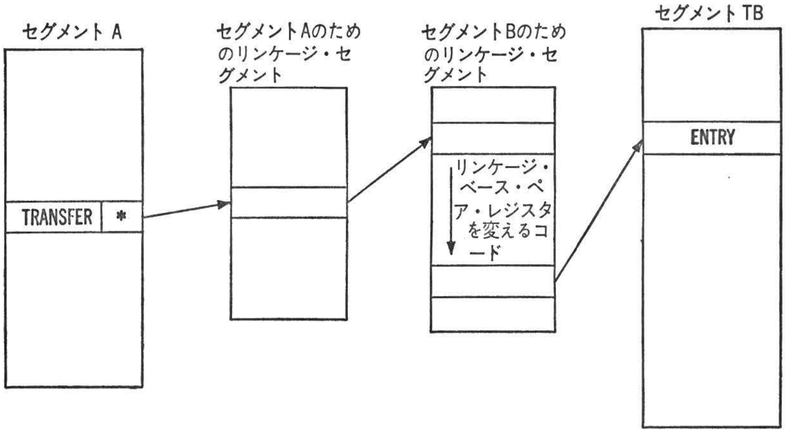


図 2.18 プログラム・セグメント間のリンケージ

の共用セグメントのリンケージ・セグメントを持っている．共用プログラムが最初に生成されるとき，リンケージ・セグメントが作られる．共用セグメントを使うそれぞれのプロセスは，システムによってこのリンケージ・セグメントの写しが得られる．これらのリンケージ・セクションは上に述べたように，適切な結合が得られるように“書き込まれる．”計算機の状態の一部としてベース・ペア・レジスタが退避されたり復帰されるので，特定な時に適切なリンケージ・セクションが参照される．

前節では，セグメント・ディスクリプタ・テーブルを使う記憶割付けによって，論理空間の記憶場所を混乱させずに，いかに多重の共用プログラムを同時に使うことができるかを示した．ここで，アドレスを持つデータ共用の問題に対する解決法を述べる必要がある．アドレスがデータ・セグメントの内部であれば，アドレスはそのセグメントの最初の場所に対して相対的に見える．データ・セグメントの最初の場所はセグメント・ディスクリプタで与えられるので，この場合は論理空間の位置付けに混乱を起こすことはない．もしアドレスがプログラム・セグメントを参照していれば，共用データを使うそれぞれのプロセスに対して，リンケージ・セクションが設定される．このリンケージ・セクションはそのデータを使うデータ・セグメントまたはプログラム・セグメントの位置付けにおいて混乱をおこさないことを保証している．これで Multics のセグメンテーション方式の概要を述べ尽くしたことになる．

2.5.8 セグメンテーション方式に対する他の方法

セグメンテーションの概念に若干の変形があることを知るために，他の方式について簡単に調べてみよう．よく知られている初期のセグメンテーションの方式は Burroughs B-5000 に見られる．B-5000 においては，プロセスは1024セグメントまで含むことができ，プログラム・セグメントは任意の長さで良い．またデータ・セグメントは1024語に制限されている．より大きな空間を必要とするデータ構造は，セグメントを複数個使う．セグメントはページ化されておらず，したがってそれが物理記憶割付けの単位になる．B-5000 システムではセグメント・ディスクリプタ・テーブルに等価なテーブルを**プログラム参照テーブル**（program reference table）と呼んでいる．全く一般性を失わずにセグメンテーションを使うことは考えていない[10]．

Multics の方式と同じようにセグメントが全論理アドレス空間（あらゆるセグメント）を原理的に参照できるようなセグメンテーションの方式では，その実現においてアドレス情報に多数のビットが必要になり，その結果多くのベース・ペア・レジスタや対になった間接語などが必要になる．実際には，どんなセグメントも限られた数のセグメントとしかコミュニケーションしない．Evans および Le Clerc のセグメンテーション方式[13] は，どんなセグメントも少数の他のセグメントしか直接アドレスしないように制限している．しかしながら，各々のセグメントはそれぞれ他のセグメントをアドレスすることができるので，全体のアドレス空間は非常に大きいにもかかわらず，必要なアドレス方式と保護のためのハードウェアは簡単にすることができる．

Scientific Control Corporation (SCC) 6700 システムは，California 大学 Berkeley で，M. W. Pirtle, B. W. Lampson, W. W. Lichtenberger らによって SCC とともに開発されたものであるが，いくつかの興味ある構想をともなったセグメンテーション・システムを備えている[16]．SCC-6700 の保護方式は4.1.2節および4.1.3節で論じる．興味ある新しい主な構想は，ハードウェアの連想マップの更新に関するものである．SCC-6700 は，プロセスにそれぞれが16K語の大きさのセグメントを32個までアクセスすることを許している．GE-645 と同じような連想マップを持っている．そのマップにないセグメントおよびページが参照されると，セグメントおよびページ・テーブルの内容で連想マップを更新するためのソフトウェア・ルーチンに制御を移すのは時間がかかるので，その代わりに特殊なマイクロプログラム制御（4.3 節のマイクロプログラミング参照）のプロセッサを呼び出し，これらの機能を行なわ

せる．この特殊プロセッサは記憶装置へのアクセスができ，特殊な設計になっているので性能が良く，通常のシステム・ルーチンよりずっと速くセグメントおよびページ・テーブルの内容で連想マップを更新することができる．SCC-6700 では命令およびオペランドの読み出しは重複して行なわれ，連想マップは多くの場合，中央のプロセッサを遅らせることなく更新できる．セグメント間のリンクは Multics と同じように間接アドレス方式によって取扱われる．

Multics 以外のシステムで専門家の注目を集めているのは Michigan 大学で開発した方式[11]に基づいた IBM 360/67[7] である．RCA Spectra 70/46 のセグメンテーション・システムも非常に似ている[9]．GE-645 と IBM 360/67 との主な相違点は，一般アドレス（セグメント番号／セグメント内記憶場所）を形成する方法にある．

IBM 360/67 の一般アドレスは，すべて普通のインデックス変更および間接アドレスを使って形成される．完全に変更が終わった後のアドレスは，セグメント番号およびセグメント内記憶場所番号の 2 つの部分に分けられる．（GE-645 と同じようにセグメント内記憶場所番号はページ化されているが詳細については省略する）．GE-645 ではセグメント番号はインデックス変更できない．このような 2 つのセグメンテーションをそれぞれ線形（linear）形式，および記号（symbolic）形式であると呼ぶ[2]．記号セグメンテーション（GE-645）では，セグメント間はセグメントの境界には連続したアドレスがないという意味で論理的に無関係である．すなわち，セグメント n の最後のアドレスとセグメント $n+1$ の最初のアドレスには何も関係が存在しない．セグメント n のアドレスからセグメント $n+1$ のアドレスを得るためにインデックス変更のアドレスを使うわけにはいかない．線形セグメンテーション（IBM 360/67）は，セグメントの境界でアドレスは連続であり，セグメント n の最後のアドレスはセグメント $n+1$ の最初のアドレスにつながっている．すなわち，セグメント n 内のアドレスを変更し，セグメント $n+1$ 内のアドレスを作り出すことができる．

この相違はいろいろな意味がある．比較的重要なものの 1 つは次のものである．線形セグメンテーション・システムでは，セグメント番号はセグメント・ディスクリプタ・テーブル中でのセグメント・ディスクリプタの相対的な位置であるので，ディスクリプタのセグメント・ディスクリプタ・テーブル内の位置が非常に重要になる．線形セグメンテーションされた空間は 2 次元ではなく，大きな 1 次元の空間である．

IBM 360/67 の 24 ビット・アドレスでは，16 セグメントしかアドレスでき

ない．このことは一般に同じセグメント内にいくつかの独立のプログラムを詰め込む必要性を生じさせている．このような使い方をすると，厳密に実際のページ化したシステムよりも少ない数のページ・テーブルで済む．なぜならば，共用プログラムのページ・テーブルは共用することができるからである．しかしながら，セグメンテーションの論理的な一般性は制約を受けるように思われる．すなわち，このように使われるとセグメンテーションは物理記憶の利用を容易にするが，厳密には論理的な概念を殺してしまうことになる．したがって，その存在の正当性が失われてしまう．

GE-645 の場合と同じように IBM 360/67 は連想マップを用いてセグメントおよびページ・ディスクリプタ・テーブルにおける2重のテーブル参照に必要な回数を減らそうとしている．しかしながら，IBM 360/67 では命令の取出しはマップだけでは行なえず，プログラム・カウンタに対するベース・レジスタのように働く別の連想レジスタを介して行なれる．それゆえ，ページの境界にまたがったループを実行する場合は，主記憶内のマップを参照する必要がある．GE-645 では，ループのページも連想マップにエントリーがあるのでこのような状況は生じない．360/67 のもう1つの特徴は，記憶保護が物理記憶のブロックに基づいており，GE-645 のようにセグメントのような論理ユニットに基づいたものではないことである．

2.5.9　まとめと結論

セグメンテーションは大きな論理アドレス空間を得るための手法であり，プログラムおよびデータの共用に関して完全な一般性を得るためにも使うことができる．この手法は2次元アドレス，記号セグメント名およびセグメント内記号場所名に基づいている．セグメンテーションは論理空間の割付けのための手法でもある．セグメントは物理空間の割付けを助けるためにページ化することができる．ページングは物理空間の割付けをするための手法である．

セグメンテーションを実現するためには，いくつかのテーブルと機構が必要である．中核のテーブルはセグメント・ディスクリプタ・テーブルである．プロセスにおいてセグメントの論理空間の記憶場所は，セグメント・ディスクリプタ・テーブル内のデスクリプタの記憶場所で表わされる．セグメント番号はセグメントに対する適当なセグメント・ディスクリプタを位置付ける．同じセグメントを共用する異なったプロセス同志は，そのセグメントを異なったセグメント番号で参照することができる．

Multics のセグメンテーション方式にとって基本的に重要な概念の1つは，

リンケージ・セクションの概念と間接アドレス方式の利用法である．リンケージ・セクションはセグメント外の参照に対して記号セグメント名とセグメント番号の間を結合する役割を実質的に果たす．リンケージ・セクションはプログラムおよびデータの共用に対する完全な一般性を得るために用いる．

セグメンテーションは大きな論理アドレス空間の割付けにおいて，完全な一般性を得るために開発されたものである．セグメンテーションの評価は次の2つのレベルで行なわなければならない．(1)基本的な仮定のレベル，すなわち，大きな論理空間を作成したり取り扱う際の完全な一般性が実際に必要である．(2)特定の実現方法のレベル，すなわち，考えている実現方法がその目標を達成する最も効率的な方法であることを保証する．

セグメンテーションは，計算機の構成に改革をもたらす重要なステップであるように思われる．主記憶よりも大きなアドレス空間を生成し，プログラマから情報がどこに格納されているかを管理する直接的な責任を取り除く必要が多くの応用分野で増えてきている．また，一般化した共用の機構は多くの応用システムの開発に要する費用を軽減し，保守を容易にする上で重要な助けになると思われる．しかしながら，ほとんどの問題が大きな論理空間を扱う際に，完全な一般性を要求しているわけではないことも事実である．そのような問題では，ベース・レジスタやページングを使って特別に設計したシステムの方が，一般的にセグメント化されたシステムよりもずっと経済的にその必要性を満たすことが多い．

セグメンテーション方式に関しては，別のリソース割付け方式をハードウェアで実現する余地があるように思われる．上述したSCC-6700で用いた特殊マイクロプロセッサは連想マップを更新するのを助けているが，それはこのような機構の1つである．Multics や IBM 360/67 のような試験的に開発された重要なシステムに関して測定をしたり，経験をつめば，有用と思われる付加ハードウェア機構の方式が疑いなく示唆されるであろう．

2.6 I/O プロセッサとモニタ・アドレスの形成

2.6.1 I/O アドレスのマッピング

アドレス方式の一面として述べなければならないのは，I/O 機器に関するものである．ページングやセグメンテーションを利用したシステムにおいて，プロセスが主記憶に対してある情報を読み書きしたいとき，プロセスは論理アド

レスを用いる．したがって，I/O プロセッサもまた，ある種のマップを介して記憶装置をアクセスすることになろう[17]．このような要求は次のようなブロック転送の場合に特に重要である．すなわち，ブロックの一部があるページの中にあり，その残りが他のページまたはそれ以上のページにまたがっている場合である．普通の I/O プロセッサでは，転送の先頭アドレスと，転送すべき語数は CPU によって I/O プロセッサに与えられ，指定されたブロックの情報が転送される．I/O プロセッサ内でマッピングしないページング・システムではCPU がページの境界を過ぎてから，I/O プロセッサに新しいアドレスを送らなければならない．この処理はI/Oの動作速度を低下させ，高速ドラムのような機器に対して，転送時に情報の消失をまねく．

XDS-940 に関しては，転送が始まる前に**直接アクセス・コミュニケーション・チャネル** (direct-access communication channel) (DACC) に 2 つのページ番号が送られる．したがって，ページの境界を横切っても 2 つ目のページ番号が転送を制御するために自動的に用いられる．その時，CPU には次のページ番号を送るように信号を出す．CPU は，1 ページの転送に対応する応答時間をもっている[18]．940 の記憶装置に直接接続される CRT・ディスプレイ・プロセッサに関しては別の方法がとられている[19]．この機器では，CPU と全く同じ記憶マップが用いられている．

セグメント化したシステムにこの考え方を適用すれば，各プロセッサ，すなわち I/O プロセッサおよび CPU，に完全なセグメンテーション・ハードウェアを必要とする．あるいは，すべてのプロセッサがそのようなハードウェアを共用できるようなある種の機構が必要になる[17]．その機構は CPU によって最初に絶対アドレスに変換するのではなく，記号セグメント名-セグメント内記号記憶場所名をI/Oプロセッサに渡すことができるものである．動的再配置のシステムの中にすべての種類のプロセッサに対して統一した経済的なアドレス形成方式を開発するにはさらに研究が必要で，これ以上コメントを述べる段階ではない．

2.6.2　モニタ・アドレスのマッピング

残りの話題は，システム・ルーチンがいかに記憶装置をアクセスするかという問題である．XDS-940 システムでは実際に，2 つのハードウェアの記憶マップがある．1 つはユーザ・プロセスのためにあり，他はシステムのためにある．システム・ルーチンが記憶装置の決められた固定的な位置を占めていれば，システムはマッピングを必要としない．しかしながら，システム・ルーチンの中にはあまり使われないもの，あるいは，十分速い応答が要求され，主記

憶に常駐しなければならないものもある．これらのルーチンへの記憶装置の割付けは必要に応じて動的に行なえることが望ましい．XDS-940 システムは，この機能をシステム・マップを使って達成している．計算機の動作しているモード，すなわちシステム・モードあるいはユーザ・モードによって2つのマップの内の1つを使う（4.1.3節でさらに述べる）．2つのマップを使うと，システムはユーザのマップに変えることなく，その機能を果たすことができる．

Multics システムは別の方法を用いている．各プロセスが持つセグメント・テーブルにシステム・セグメントを割当て，論理的に各プロセスの一部に見たてている．この方法で難点と思われることは，ユーザ・プロセス実行中におこるシステム割込み，およびユーザからのシステム呼び出しで連想マップを変えることである．したがって，ユーザ・プロセスに制御が戻される時にそのマップを復帰させるために，2重のテーブル参照をしばしば行なう必要がでてくる．文献ではこの点に関して何のコメントも与えていない．この問題が潜在的にきびしいものであるかどうかは，システムへの割込みおよび飛び越しの頻度にかかっている．この問題に関しては出版され，入手可能な文献はほとんどない．Freibergs[20] がマルチ・プログラムのバッチ・システムにおけるシステムに対する呼び出しは，一般に命令を100ないし1000個実行するごとに起こるといっているだけである．タイムシェアリング・システムには別の機能が付加されるので，システムへの呼び出しの比率は比較的高いと予想される．将来のシステムを設計する際に有用な統計量の収集については，第7章で論じる．

2.7 ま と め

本章ではアドレス方式および記憶装置の割付け，物理記憶空間，および論理記憶空間に関連する重要な概念をいくつか紹介した．物理記憶空間と仮想，すなわち論理空間の区別は，タイムシェアリング・システムの設計には非常に重要になっている．ユーザおよびシステム・プロセスは，それ自身の論理記憶空間を持つ仮想プロセッサ上で実行するものと見なすことができる，

ベース・レジスタおよびページングの概念は，同一の1次元形式の論理空間を作り出し，物理空間の割付けを助ける方式である．この2つの方式はプログラムやデータを完全な一般性を損わずに共用しようとすると，論理空間の割付け問題におちいる．論理空間割付けの問題を避けるために開発された方式のセグメンテーションは，2次元の論理空間を作り出す．このような概念の動機について，若干詳細に調べた．

その他，論理空間から物理空間にマップする考え方を主に紹介した．マップはハードウェアまたはソフトウェア，あるいは（通常）両者を組合わせて実現できる．

参　考　文　献

1 **McGee, W. C.:** On Dynamic Program Relocation, *IBM Systems J.*, vol. 4, no. 3, p. 184, 1965.

2 **Randell, B., and C. J. Kuehner:** Dynamic Storage Allocation Systems, *Commun. ACM*, vol. 11, no. 5, p. 297, May, 1968.

3 **Corbato, F. J., et al.,** An Experimental Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 21, pp. 335–344, 1962.

4 **Amdahl, G.:** Unpublished simulation study, IBM Corporation, 1965.

5 **Kilburn, T., et al.:** One-level Storage System, *IEEE Trans.*, vol. EC-11, no. 2, p. 223, 1962.

6 **Dennis, J. R.:** Segmentation and the Design of Multiprogrammed Computer Systems, *J. ACM*, vol. 12, no. 4, pp. 589–602, October, 1965.

7 **Gibson, C. T.:** Time-sharing in the IBM System/360: Model 67, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 28, p. 61, 1966.

8 **Glaser, E. L., et al.:** System Design of a Computer for Time-sharing Applications, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 197–202, 1965.

9 **Oppenheimer, G., and N. Weizer:** Resource Management for a Medium Scale Time Sharing Operating System, *Commun. ACM*, vol. 11, no. 5, p. 313, May, 1968.

10 **Wald, B.:** "The Descriptor: A Definition of the B5000 Information Processing System," Burroughs Corp., Detroit, Mich., 1961.

11 **Arden, B. W., et al.:** Program and Addressing Structure in a Time-sharing Environment, *J. ACM*, pp. 1–16, January, 1966.

12 **Daley, R. C., and J. B. Dennis:** Virtual Memory Processes, and Sharing in MULTICS, *Commun. ACM*, vol. 11, no. 5, p. 306, May, 1968.

13 **Evans, D. C., and J. Y. LeClerc:** Address Mapping and the Control of Access in an Interactive Computer, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 23–30, 1967.

14 **Graham, R. M.:** Protection in an Information Processing Utility, *Commun. ACM*, vol. 11, no. 5, p. 365, May, 1968.

15 **Organick, E. I.:** A Guide to Multics for Subsystem Writers, *Project MAC Doc.*, March, 1967.

16 "6700 Time-sharing Computer Reference Manual," Scientific Control Corporation, Carrollten, Texas, February, 1969 (additional information also obtained at a public presentation of the 6700).

17 **Smith, Arthur A.:** Input/Output in Time-shared, Segmented, Multiprocessor Systems, (thesis) MAC-TR-28 *MIT Project MAC Doc.*, June, 1966.

18 Pirtle, M. W.: Modifications of the SDS-930 for the Implementation of Timesharing, *Univ. Calif. Berkeley Project Genie Doc.*, M-1, April 4, 1967.

19 Watson, R. W., et al.: A Display Processor Design, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 35, 1969.

20 Freibergs, I. F.: The Dynamic Behavior of Programs, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1163–1168, 1968.

21 Knuth, D. E.: "Fundamental Algorithms: The Art of Computer Programming," Addison-Wesley Publishing Company, Inc., Reading, Mass., 1968.

22 Conti, C. J.: Structural Aspects of the System 1360 Model 85, *IBM Systems J.*, vol. 7, no. 1, pp. 2–29, 1968.

23 Wegner, P.: "Programming Languages, Information Structures and Machine Organization," McGraw-Hill Book Company, New York, 1968.

24 Corbato, F. J.: System Requirements for Multiple Access Timeshared Computers, Project MAC Document MAC-TR-3.

第 3 章

コミュニケーション

3.1 コミュニケーションの問題

3.1.1 概　　要

タイムシェアリング計算機システムの目的は，ユーザの要求を満たすために計算機システムのリソースを高速に時分割することである．システムは，ユーザが望むように仕事を終え，ユーザが望む応答性を満足し，またシステム・リソースを効率よく利用するためにこの多重化を行なう．これらのシステムについて有用な見方は，接続されている数多くの，そして多種多様な機器間における情報の転送と変換を制御する大きな情報交換センタと見ることである．

3.1.2 主記憶とのコミュニケーション

現在の計算機機構においては，情報がある機器から他の機器に移動するときに情報が通過する中心部分は主記憶である（情報を変換する場合に CPU に寄り道することもある）．主記憶は主なシステム・リソースであり，その結果，コミュニケーションにおける問題が起こり得る所である．タイムシェアリング・システムでは，複数の CPU，高速転送の 2 次記憶および多数の I/O 機器が主記憶へのアクセスを共用する．2 次記憶や I/O 機器を制御し，主記憶とのコミュニケーションを行なうプロセッサは，普通，チャネル，I/O 制御装置，あるいは，I/O プロセッサと呼ばれる．設計上の問題点は種々のプロセッサと主記憶間で十分なデータ転送能力を持たせることである．主記憶はある特定の時点でデータを受取ったり送ったりし，またプロセッサと主記憶間の転送 - アドレッシング経路がデータ転送を実行するが，理想的にはこれらの能力に関係なく各プロセッサが都合の良い時点で主記憶との間でデータを転送できるようにしたい．残念ながら，経済性や技術的問題を考慮すると，このような理想的な

考えは現実のシステムには適用できない．主記憶とのコミュニケーションに関連した概念は，3.2 節で論じる．

3.1.3 補助記憶および I/O 機器とのコミュニケーション

補助記憶や I/O 機器との基本的なコミュニケーションの問題は，主記憶への直接転送経路のアクセスを得ることである．タイムシェアリング・システムに接続される機器には種々なものがある．これらの機器に関係するものはデータ転送速度の限界である．主記憶への直接転送経路には，記憶モジュールのアクセスの競合を解決する論理回路や経路の両端に送受信回路が必要である．そのために，各機器に別々の経路を設けることは不経済である．しかしながら，接続される機器の転送速度に限界があることから，多数の機器を1つの主記憶への直接転送経路を同時に共用させる I/O プロセッサを設計することが可能である．この問題は3.3節で論じる．データ転送に関する要求事項の他に補助記憶と I/O 機器に対する制御のコミュニケーションに関する要求事項もある．(3.3 節および4.2 節参照)

3.1.4 遠隔機器とのコミュニケーション

遠隔機器に関係して，3つの主要なコミュニケーションの問題がある：

1. 中央の装置と遠隔機器間での情報転送
2. 通信回線と中央の装置間のインタフェース
3. 通信回線と遠隔機器間のインタフェース

本書ではタイムシェアリング・システム設計者が直面する論理的な問題や概念についてのみ扱い，その他，一般に電子工学者に興味ある多くの信号転送やインタフェースの問題については触れない．情報の転送に関しては，標準の電話回線をディジタル情報に使う技術や，複数の機器で回線を共用させる技術，および，遠隔地点間のコミュニケーションの同期をとる技術について考えなければならない．通信回線と中央装置間のインタフェースに関しては，コミュニケーションを行なっている機器の識別，制御，およびアドレス方式の問題や，また，転送された情報を中央計算機が処理できる形に変形する問題等がある．通信回線と遠隔機器間のインタフェースに関しては，情報のエンコーディングやデコーディング，そして識別を行なう問題がある．これらの問題については3.4節で論じる．

3.2 主記憶に関するコミュニケーション

3.2.1 多重記憶ボックスとバス構成

記憶コミュニケーション・システムの設計で解決すべき技術的問題は，主記憶とアクセスを要求するすべてのプロセッサ間に十分な転送能力を持たすことである．実際のシステムでは，プロセッサと主記憶間でデータを転送できる速度は，記憶装置自体と記憶バスの転送能力により制限される．主記憶が情報を転送できる速度は，しばしば，**記憶バンド幅**（memory bandwidth）と呼ばれ，通常毎秒当たりの語数で測られる．バンド幅の限界はまた記憶バスにも存在する．記憶システムは数台のプロセッサで共用されるため，設計の際には，記憶バスや記憶装置のような装置を数台のプロセッサが同時に利用する際に生ずる**競合**（interference）による重大な性能の低下をもたらさないように，設計上の配慮がされねばならない．図 3.1 はリソース共用システムにおける一般的な記憶装置の構成の方法を示している．

記憶装置を，1 セットのアドレッシング回路と 1 セットの読出し／書込み回路の一本化したユニットと考えるより，各々1 セットのアドレッシング回路と読出し／書込み回路を持った m 個の小さなボックス（box）に分割して考える．たとえば，XDS-940 では 64K の記憶装置は 4 個の 16K ボックスに分けられている．各ボックスをアクセスするのは n 個までのデータ・アドレス・バスである．各々のデータ・アドレス・バスは，データや要求されたアドレスのために記憶装置間にケーブルを持っている．n の値は現在のシステムでは多数の機器がバスを共用でき，プロセッサの数が限られ，記憶バンド幅に限界があるため，普通 2 から 5 までの値をとる．必要なバスの数とバス構成に関しては，

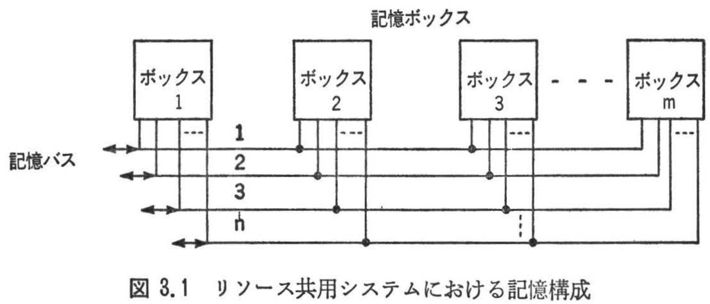


図 3.1 リソース共用システムにおける記憶構成

さらに3.3節で論じる．1つの代表的な記憶ボックスの図が，図3.2に示されている．バス1，2，3，あるいは4のどの組合わせでも，記憶ボックスに同時にアクセスを要求できる．アクセスの優先制御は特定の記憶サイクルでどのバスにアクセスを与えるかを決定しなければならない．そして，その他のバスの要求は次のサイクルまで待たされるか，あるいは，取り消され，この場合は次のサイクルで再び要求を出す必要がある．**バス構成**（bus structure）という用語は，しばしば，バス優先制御系を指すのに使われる．記憶システムのバンド幅は，それに関係する2つの限界値を持つ．

図3.1に示したシステムに対する最大の記憶システム・バンド幅は$p \times R$で与えられる．ここで，pは記憶モジュール数mとアクセス経路数nの小さい方の値であり，Rは各ボックスの最大転送速度である．言い換えると，最大転送速度は各経路が別々のモジュールにアクセスを要求する時に得られる．

最小転送速度は正にRであり，すべての経路が同じモジュールにアクセスを要求する時に生じる．この場合，**競合**が生ずる．

図3.1で示した方式では，1つ以上のボックスに同時にアクセスを許して競合を取り除いている．もしバス2がボックス3にアクセスを要求すると同時に，バス1がボックス2にアクセスを要求した時，各ボックスは自分自身のアドレッシングおよび読出し／書込み回線を持っているので，両方の要求が許可される．図3.1に示された方式をもってしても，記憶アドレスがボックス内で連続的になっている時には重大な競合が起こりうる．たとえば，ボックス1が1から16Kのアドレスを持ち，ボックス2が16K+1から32Kのアドレスを持つ場合である．主記憶の速度で転送する高速ドラム・プロセッサの場合を考えてみよう．この機器が演算ユニットよりも主記憶へのアクセスが高い優先順位

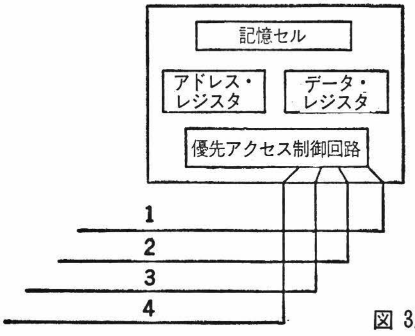


図3.2　記憶ボックスの詳細

を持つとすれば，ブロック転送中に演算ユニットがドラム・プロセッサに使われている記憶ボックスをアクセスしようとするときには長い間そのアクセスを拒否される．この問題を回避するために，設計者は**インタリービング**（interleaving）と呼ばれる技術を開発した．

インタリーブされた記憶装置では，連続したアドレスは別々の記憶ボックスに置かれる．たとえば，2記憶ボックス・システムでは，すべての偶数アドレスは1つのボックスにあり，すべての奇数アドレスはもう一方のボックスに置かれる．インタリーブされた記憶装置を使うと，1台のプロセッサが記憶装置を長い時間拘束する確率を著しく減少できる．設計上の問題は，各ボックスの大きさを決定することと，すべてのボックスに渡って，あるいは，ボックスのグループに対してインタリーブを行なうかどうかである．これらの疑問はシミュレーション手法で検討できる[1].

このような多重バス構成の一方式で記憶参照を行なう際は，まずプロセッサは記憶バスを使用するための要求を出し，この要求が受け入れられる場合は，このバスを使って該当する記憶モジュールへ記憶参照要求を送る．もしどちらか一方の要求が拒否された時にはプロセッサはこの手順をくり返すか，その要求を保留しておく．

記憶バスあるいは記憶ボックスに対する要求の競合を解決するために，ある種の優先機構が必要である．優先権は代表的にプロセッサおよび記憶バスに与えられる．後者の場合は，記憶モジュールに対する要求の競合を解決するために使われる．優先方式を含む全記憶構成の設計は，この重要な点における障害を取り除き，新しい I/O 機器やプロセッサの付加に対しても拡張性を持たせようとすれば難しくなる．

3.2.2 記憶バス構成の限界

先に述べた記憶構成に重点を置いて，タイムシェアリング・システムに用いる場合に要求を十分満足できない記憶構成について論じよう．図3.3に記憶アクセス経路の様子が示されている．この構成は多数の小中規模計算機に一般に見られるものである．記憶装置はインタリーブされておらず，CPU と I/O プロセッサが共用する記憶装置へのアクセスおよびアドレッシング経路はただ1つだけしかない．このようなシステムでは，もし I/O プロセッサが必要な時に記憶装置をアクセスできないとデータを失うことになるので，通常，I/O プロセッサは記憶装置のアクセスに対し，CPU よりも高い優先順位を持っている．I/O プロセッサが記憶装置のアクセスを要求する時，アクセスとアドレ

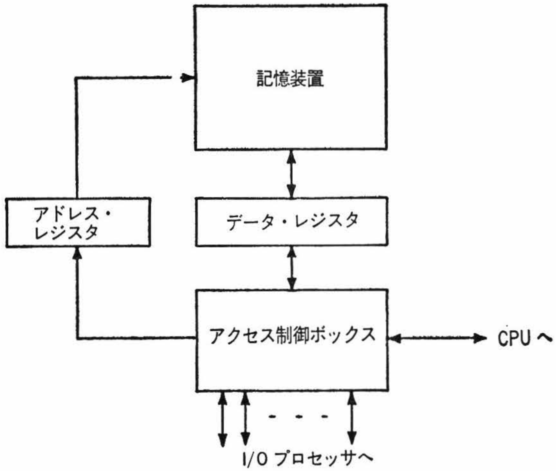


図 3.3 単一記憶アクセス経路

ッシングの間 CPU は停止している．そのような記憶構成では CPU との重大な競合が生ずる可能性がある．

多くの計算機に使われているこの種のプロセッサは，**マルチプレクサ・プロセッサ** (multiplexor processor) と呼ばれる．それは複数の低速 I/O 機器を同時にデータ経路に接続できるようにする．低速 I/O 機器から送られる字を語に組合わせるハードウェアは CPU のレジスタを使う．さらに，低速 I/O 機器のデータ転送に関係した記憶アドレスと部分的に組合わされた語のための一時的な記憶場所には主記憶領域の一部が使われる．その結果，低速 I/O 機器からの字転送中に CPU の競合が加わる．

3.2.3 同期式および非同期式コミュニケーション

記憶システムは，**同期式** (synchronous) と**非同期式** (asynchronous) の 2 つの形式に分類できる．同期式のシステムでは，記憶装置に対する要求は離散的な時点でのみ出すことができ，**同時要求** (simultaneous request) の概念が使われる．記憶要求は，通常，記憶サイクルの開始の直前に出され，この時点に記憶参照を要求しているすべてのプロセッサは，その要求を要求線上にのせ，優先順位機構はそれらの間の仲裁をする．非同期式のシステムではプロセッサはいつでも記憶参照要求を出すことができ，同時要求の考えはなくなっている．GE-645 システムでは非同期式記憶アクセスが使われており，一方，

XDS-940 は同期式を使ったシステムの一例である.

非同期式の記憶参照を用い，主要なシステム・モジュール間で非同期式コミュニケーションを用いる理由は，システム・モジュールを異なった性能を持ったものに置き換えることができ，しかも置き換えないモジュールはそのまま変更しなくても良いからである．また，非同期式の場合は最大速度でコミュニケーションできるが，同期式システムではある一定の時間に制限されてしまう．非同期式システムにおいては，優先順位の概念はアクセスの要求が一般に先着順サービスで取扱われるために殆んど意味がない．それゆえ，種々のプロセッサの物理的，統計的性質は記憶バスを共用させ，あるいは，**可変優先要求**（variable-priority request）を付加することによって，与えられた価格での性能を改善する利益は容易には得られない．同期式システムでは，可変優先情報を有効に使うことができる．

3.2.4 Multics システムの記憶バス構成

典型的な非同期式システムが 図 3.4 に示されているが，そこでは，各々の CPU が独自の各記憶ボックスへのバスを持っている．スワッピングに使われる高速2次記憶機器（ドラム）は，各記憶モジュール独自のバスを持っていることに注意されたい．最後に GIOC（general I/O controller）と呼ばれるプロセッサは，各モジュールへ別々のバスを持っていることも注目してほしい．（GIOC については，さらに3.3.2節で論じる．）その主要な機能の1つは，そのバスと記憶モジュールへのアクセスをスケジュールすることである．システムは非同期式であるため，記憶モジュールへのアクセスは先着順サービスを基にしている．記憶モジュールはインタリーブされている．この非同期方式はハードウェアやソフトウェアの修正を少しで良いか，または全く行なわないで簡単により高速の CPU や記憶装置に交換できる．非同期式システムで最大のバンド幅を得る際に生ずる1つの問題は，連続動作している記憶装置は本質的に同期式と同じものであり，そのためにアクセスの判断は一定の時点で行なわれなければならず，たとえ高い優先順位の要求が若干遅れて来たとしても簡単に取り消すわけにはいかないことである．

3.2.5 XDS-940 の記憶バス構成

記憶装置とバスのバンド幅を増加させるための経済的で単純な優先順位機構を使った同期式システムの実例が XDS-940 であり，その概要を 図 3.5 に示す．このシステムはまた非同期式システムに利用できるその他の多くの概念を

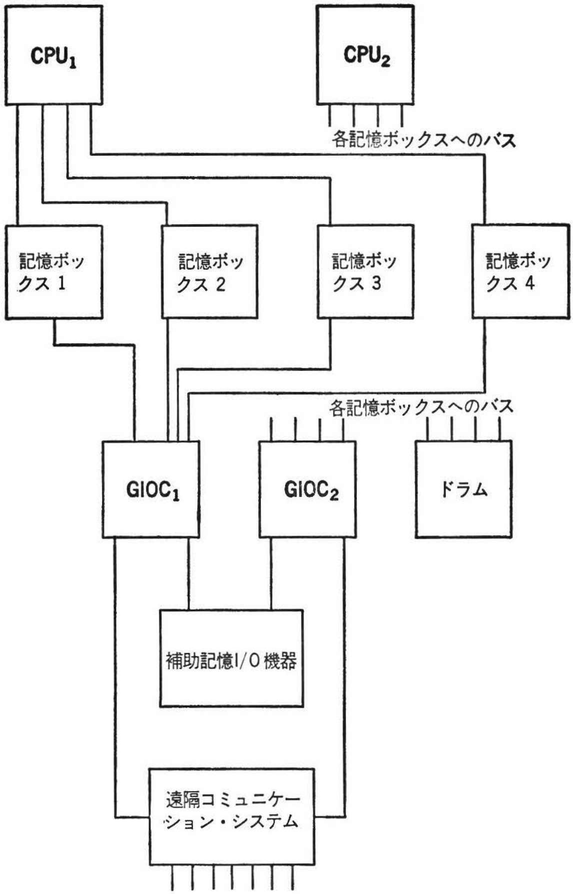


図 3.4 典型的な Multics 構成

示している.

940 の記憶装置へは 2 本のバスがある. 記憶装置は 4 重にインタリーブされている. 940 の基となった XDS-930 では, 図中で B_2 と呼ばれるバス (第2経路) がバス B_1 (第1経路) よりも高い優先順位を持っている. この優先順位の配分は第2世代および第3世代の計算機ではよく見られるが, ドラムやディスクのような高速 I/O 機器は必要な時に記憶装置へアクセスしなければならず, さもなければデータが失われているという考えに基づいている. これらの機器は回転して連続的にデータを送出しているので, データが失われる可能

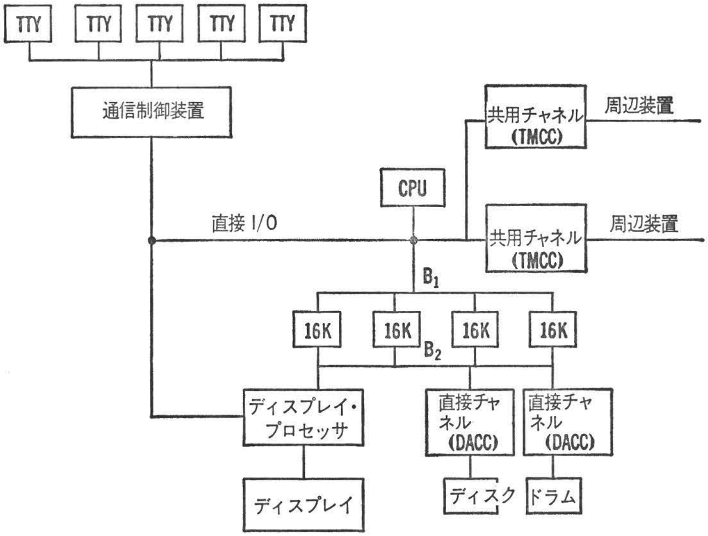


図 3.5 XDS-940 構成

性がある．930の**直接アクセス・コミュニケーション・チャネル**(direct-access communications channel) (DACC) 中には，図 3.6 に示したように1個のレジスタ・バッファがある．機器から読出し時に，**アセンブリ・レジスタ** (assembly register) は4個の6ビット字を24ビット語にパックし，その語をバッファ・レジスタに転送する．この時点で，930 DACC はバッファ・レジスタ

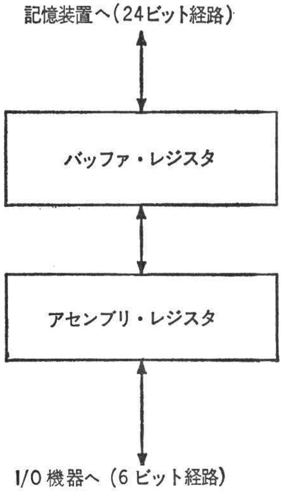


図 3.6 XDS-930 DACC

の内容を記憶装置に格納するための記憶サイクルを要求する．第2経路は第1経路より高い優先順位を持っているため，この要求は受け入れられ，バッファ・レジスタの持つ本来の価値はでてこない．機器への出力の場合は方向が逆で同様な動作が行なわれる．

アセンブリ・レジスタが1語をバッファ・レジスタに転送する時，バッファ・レジスタは少なくとも3記憶サイクルを持つ．その中で，アセンブリ・レジスタが次の1語を転送してくる前にその内容を記憶装置へ転送する．それゆえに，これら3サイクルの内の1サイクルの間に，バッファ・レジスタは転送を行なうことができる．バッファリングの目的は時間を稼ぐことである．940ではこのバッファリングの有利性を得るために，第1経路より優先順位が高くても低くても第2経路に記憶アクセス権を与えている．たとえば，アセンブリ・レジスタがその内容をバッファ・レジスタにに転送した後，DACCは低い優先順位で記憶装置へ第1要求を出す．この要求が受付けられる時には競合は生じていない．この要求が拒否された時は第2の低い優先順位の要求が出される．この第2要求が受付けられなかったならば，第3の要求が高い優先順位で出される．このCPUに対する記憶装置の競合は非常に低いことがシミューションで示されている[1].

XDS-940について終わる前に，他の計算機にも見られるような図3.5で示す直接 **I/O** (direct I/O) と名付けられた機構について指摘しておくべきであろう．直接I/OとはCPUが直接的に1つの命令を実行して，I/O機器間で1語あるいは1単位の情報を入力あるいは出力することである．中央システム内でのこの語の行先あるいは出所はCPUレジスタか記憶アドレスであり，それらは命令内で指定される．転送される各語はCPUに直接制御される．この機構はCPUにより一旦開始されるとCPUの動作に関係なく独立に情報を記憶装置間で転送するI/Oプロセッサあるいはチャネルとは対照的である．

3.2.6 バッファ記憶による記憶バント幅の拡大

記憶装置のコミュニケーションの障害を生む要因の1つとして，主記憶のアクセス時間とプロセッサの論理回路の速度の差があげられる．プロセッサの論理回路は，記憶アクセス時間よりも高速である．この問題を克服するのにしばしば使われる方法として，一度にいくつかの命令を記憶装置から取出したり，命令とオペランドの取出しを重複することが行なわれる．この方法はバッファリングを形成し，中央プロセッサに対する記憶バンド幅を拡大するのに極めて有効であったが，I/Oプロセッサに対する記憶バンド幅を同様に拡大させるに

至っていない．さらに別に考えられたバッファリングの方法は，すべてのプロセッサに対する全記憶バンド幅を拡大させることである．この方法は高速バッファ記憶を各記憶ボックス，あるいは各プロセッサに取り付けることである．この方法は最初に California 大学の M. W. Pirtle らによって開発されたシステムに使われ，そのシステムを Berkeley システムと呼んでいる．このシステムの多くの考え方が SCC-6700 や他の計算機に取り入れられている[2]．2番目の方式が IBM 360/85 にみられる[3]．現在では IBM 360/85 は CPU に対してのみバッファ記憶を取り付けているが，基本的な考えは一般化できる．目標はこの高速バッファ記憶をプロセッサに対して，すべての記憶装置が高速の半導体記憶装置であると見なせるようにすることである．各システムの比較を行なう前にそれぞれの概要を述べた．Berkeley システムについての論議は SCC-6700 についての文献[2] に基づいている．

3.2.7 IBM 360/85 の記憶構成

モデル85の記憶システムの概要を 図 3.7 に示す．このシステムの主記憶のサイクル時間は約1マイクロ秒である．500K と 1000K語（32ビット）の記憶構成では，記憶装置は4重にインタリーブされている．それより小さい構成では記憶装置は2重にインタリーブされている．バッファ記憶は CPU に対してのみ有効で，I/O プロセッサや他のプロセッサには有効でない事に注意してほしい．バッファ記憶のサイクル時間は80ナノ秒である．バッファ記憶は，4K，6K，8K語のいずれでも付けられる．このシステムの設計は，CPU から見た記憶装置の実効速度を増加させるように考えられている．すべてのプロ

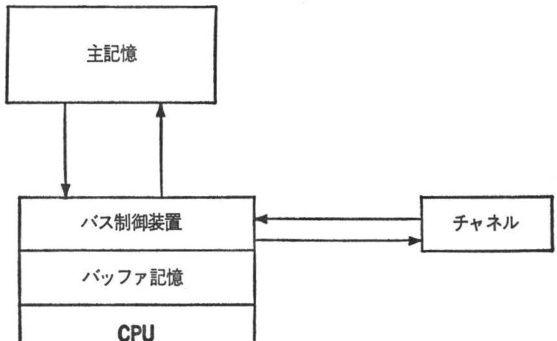


図 3.7 IBM 360/85 記憶システム

セッサと記憶装置間の高速データ転送の重要性は，この計算機では十分に生かされていない．なぜなら，われわれの見たところでは，それはまだ主にCPUを使う科学計算向のバッチ処理計算機と思われるからである．記憶バスは予定された主要な応用に対して必要なバンド幅をもたらすために，4語の幅を持っている．この構成はI/O向のシステムに対して殆んど有利性はないが，基本的な考えは拡張できる．

主記憶とバッファ記憶は256語の**セクタ**（sector）の集まりで構成されている．動作中，バッファ記憶と主記憶セクタ間は1対1の対応がとられ，各バッファ記憶セクタは1つの異なった主記憶セクタに割当てられる．バッファ記憶セクタの数に限りがあるために，主記憶セクタの大部分はバッファ記憶セクタの割当てを受けない．各バッファ記憶セクタは14ビットのセクタ•アドレス•レジスタを持ち，それは割当てられた主記憶セクタ・アドレスを保持する．

動作中にバッファ記憶セクタはプログラムに連続的に使われる主記憶セクタに割当てられるように動的に調達される．もしプログラムがバッファ記憶セクタに割当てられていない主記憶セクタから命令を取出そうとした時は，1つのバッファ記憶セクタがその主記憶セクタに割当て直される．バッファ記憶セクタを効率良く再割当てするために，セクタの順位に関する情報が**アクティビティ・リスト**（activity list）に保持される．

バッファ記憶セクタが異なった主記憶セクタに割当てられるとき，その主記憶セクタ内にあるすべての256語が一度にバッファ記憶にロードされるのではなく，各セクタは16語から成る16個のブロックに分割され，要求された時にそれらがバッファ記憶内に入れられる．

記憶装置への書込みは，常に主記憶をも更新する．変更しつつある主記憶セクタがそれに割当てられたバッファ記憶セクタを持っていれば，バッファ記憶もまた更新される．それ以外はバッファに関係した動作は行なわれない．バッファ記憶内にすべてのデータは主記憶内にも存在するため，バッファ記憶セクタの再割当て時に，バッファ記憶から主記憶へのデータの転送は必要がない．

80ナノ秒のサイクル2つが，バッファ記憶内のデータを取出すために必要である．最初のサイクルでは，データがバッファ記憶内にあるかどうかを決定するために，セクタ・アドレスと**有効ビット**（validity bit）が調べられる．2番目のサイクルはバッファ記憶からデータを取出すのに使われる．もしデータがバッファ記憶内になければ，主記憶からバッファ記憶内にブロックをロードする間に余分なサイクルが必要である．

Berkeley システムと同様に，この記憶システムを設計する際には，シミュレ

ーションが大幅に使われた. **リプレースメント(置換え)アルゴリズム**(replacement algorithm), バッファ記憶の容量, セクタとブロック容量の選定といった, 決定されなければならない多くの重要なパラメータがある[3].

代表的な科学計算向のジョブ・ミックスを実行したシミュレーションでは, 80ナノ秒の記憶装置だけで構成された理想的なシステムに比べて, このシステムの平均性能は81パーセントであることが分かった[3]. それは平均でCPU は参照する情報の81パーセントをバッファ記憶から得ていることを示している.

3.2.8 Berkeley システムの記憶構成

Berkeley システムを開発した California 大学のグループは XDS-940 を開発したグループと同じである. このシステムの設計者達は, 主および2次記憶の両方を一体化したユニットとして取扱い, すべてのプロセッサ間での競合を最小にする記憶構成が主要な問題であることを認識していた. Berkeley システムの記憶構成を図 3.8 に示す.

基本システムは, 16K, 1マイクロ秒の磁心記憶とこれから述べる高速な200ナノ秒のレジスタを6個含んだ記憶ボックスを8つ備えている. さらに記憶ボックスを増設することが可能である. 主記憶は8重にインタリーブされ, 常に48ビットの倍長語(命令と固定小数点データは24ビットである)が取出される. アドレス方式と優先順位機構のすべてを述べることは本書の範囲外であるが, いくつかの重要な特徴は述べることができよう. 各ボックスの6個の高速レジスタ群は, 倍長語, アドレス, および種々の状態情報や優先順位ビットのための記憶レジスタである.

先に同時性の概念が正確に成り立たないので非同期式システムでは複雑な優先順位機構が使えないことを述べた. しかも記憶サイクルが基本時間単位である同期式システムは, 非同期式システムより高速にならないことも述べた. こ

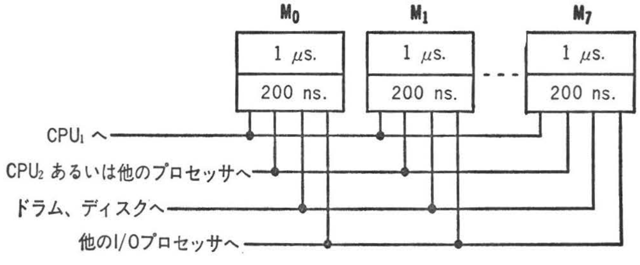


図 3.8 Berkeley システムの記憶構成

のシステムの設計者達は同期式にすることを決定したが，プロセッサに自分自身のペースで処理を行なう大きな自由を与え，200 ナノ秒の主記憶を効率的に使うために，計算機の基本的なタイミング単位として，そしてまた，記憶ボックスに要求できる時間間隔として100 ナノ秒を用いた．

このシステムには記憶ボックスのアクセスに対する優先順位と，各ボックスの磁心記憶部分に対する優先順位の2種の主な優先順位が設けられた．最初の優先順位の問題は2つの方法で扱われる．(1)主記憶に対する要求は，高い優先順位か低い優先順位かのどちらかである．(2)2台以上のプロセッサが同時に同じボックスに対して同じ優先順位でアクセスを要求したときは，バス間の配線（hard-wired）優先順位によって競合を防ぐ．配線優先順位の低いバスが高い優先順位を示してサイクルを要求したとき，たとえ配線優先順位の高いバスが低い優先順位でサイクルを要求していたとしても，先にアクセスを取る．一旦ボックスに対するアクセスを許されると，最初の100 ナノ秒間にアドレスが高速記憶内のアドレスと比較され，もしそれが取出し要求であり，状態ビットがそのデータ部分が有効なことを示しているならば，そのデータは次の 100 ナノ秒の間にプロセッサへ引き渡される．それ以外の場合は，布線化されたアルゴリズムがこの要求をレジスタにセットする．磁心記憶をアクセスする順位が低い（low）か，予告可能（warning）か，中間（medium）か，高い（high）かを示す磁心アクセス優先順位情報がその要求に付加される．この優先順位情報は高速レジスタ群に要求とともに格納され，次の磁心記憶サイクルで高速レジスタ内のどの要求を処理するかを決定するために使われる．予告可能レベルは前もって次の要求をどのボックスに出すか分かっているプロセッサが，高速レジスタを確保するために使われる．2番目の予告可能要求があるボックスに出されると，そのボックス内のすべての前に出された予告可能要求は高い優先順位にリセットされる．

CPU は命令とオペランドを先取りするように設計されているため，情報を実際に必要になるまで高速記憶内に入れておき，要求を十分後にのばすことができる．同様に，ドラムやディスク機器はブロック転送の際に，次にどのボックスが必要か前もって知ることができる．書き込みときには，プロセッサはその情報を高速記憶内に残し，後で磁心記憶を更新するために使う．書き込みした記憶場所と同じ記憶場所に対して，取出しがわずか後に生じた時は高速記憶から情報を得ることができる．小じんまりとしたループに対しては，すべての情報は高速記憶内に入っている可能性がある．この記憶構成の結果はシミュレーションによってすべてのプロセッサを最大速度で実行させた場合にプロセッ

サと記憶装置間で毎秒1500万語以上の情報が流れることが示された．そのシミュレーションから6個の高速レジスタを使用することが決定された．

上で述べた記憶構成のほかに，Berkeley システムは別の独特な設計方式がとられている．種々の I/O プロセッサは以前にはソフトウェアで行なわれていた仕事に対して CPU とは独立に考えられるようにするためにマイクロプログラム化されている．ドラム・プロセッサを例にとると，以前には CPUに含まれていたスワッピング機能をドラム・プロセッサで取扱うことができる．多重プロセッサについての考え方の有用な方向は，多重プロセッサが種々の記憶媒体による階層に良く似た1つの階層を形作っていると見なすことである．このようなプロセッサの例は，6.1 節で述べる．

われわれの観点では，360/85 よりも Berkeley システムの記憶装置におけるバッファ構成の良い点は，高速記憶が CPU だけでなくすべてのプロセッサに対しても有効なことである．さらに，Berkeley システムでは書き込み動作は高速記憶に対して行なわれ，低速の磁心記憶には行なわれない．タイムシェアリング・システムでは主記憶と2次記憶間の情報転送はたびたび行なわれるので，Berkeley システムのシステム方式は特に重要である．Berkeleyシステムの機構はおそらくあまりハードウェアを必要とせず，そのためあまり高価ではないだろう．Berkeley システムの同期方式によって，磁心記憶の要求に対する競合を減らすあらゆる情報を使う優先順位方式を開発することが可能となる．前に述べたように，360/85の方式は一般化すれば上述したいろいろな欠点を除くことができる．

3.2.9 まとめ

記憶バンド幅を拡大させることに関した多くの概念を紹介した．それは記憶ボックスの多重化，インタリーブ方式，そして高速バッファ記憶であった．記憶バンド幅を拡大する方法としての同期式および非同期式コミュニケーションの利点と欠点についても述べた．設計者には記憶ボックスの大きさ，記憶バスの数およびコミュニケーション形式の選択を含む設計上のつり合い問題があることがわかった．高速バッファ記憶を使う場合には，決定しなければならない設計パラメータの数はさらに多くなる．過去のシステムの性能における十分な統計資料なしにはこれらの設計上の決定を行なうことは困難であり，おそらくシミュレーション実験の助けが必要になるであろう．

3.3 補助記憶および I/O に関するコミュニケーション

3.3.1 n 個の機器を m 個の記憶モジュールに接続する方法

直接接続　補助記憶と I/O 機器についての基本的な問題は，主記憶への直接転送経路のアクセスを得ることである．さらに，主記憶内のどこから，あるいはどこへ転送するかを決める論理回路や，どの位の量の情報を転送するかを決める論理回路が必要である．理論的にこれらの問題は各 I/O 機器から各記憶モジュールに直接転送経路，ならびにアドレスおよび計数論理回路を持たすことにより解決される．このような方法を 図 3.9 に示す．この方法は線路選択論理回路や駆動回路の点からみた価格を考えると満足できない．その価格を減らすためには，I/O 機器や記憶装置の情報転送速度に関してわれわれに得られる情報を使う必要がある．第 3.9 図を見ると，1 つの記憶モジュールは一度にただ 1 つの I/O 機器に対してのみ情報を転送することができ，I/O 機器

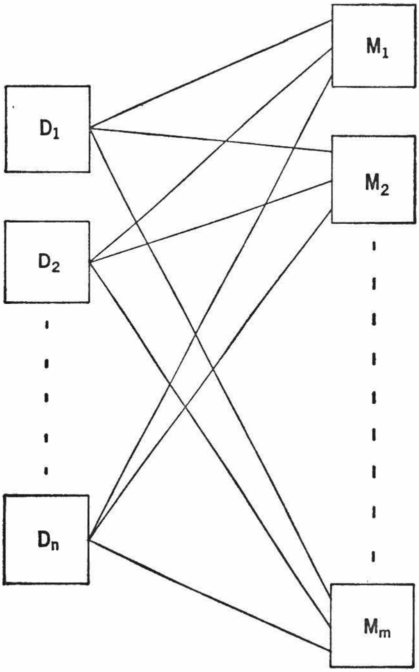


図 3.9　n 個の機器と m 個の記憶モジュールとの直接コミュニケーション

は一時にただ1つの記憶モジュールへだけしか情報を転送できないことが分かる．n または m のどちらか大きい方で決まる線路でしか転送は起こり得ないが，実際には $n \times m$ 本の線路がある．それゆえ，大部分のアクセス線路は使われず，これらの線路に関する論理回路と駆動回路は効率的には使われていない．

クロスバ接続　前の知識を使って，各 I/O 機器にどのアクセス線路を使うかを決定する論理と，各記憶モジュールにアクセス要求の競合を仲裁する論理を付加することができ，それを図 3.10 で示したように相互接続して中央にまとめることができる．この方法で I/O 機器と記憶モジュールの間の経路を決めるのに必要な論理を**クロスバ・スイッチ** (crossbar switch) という．**スイッチ・ポイント**という円で囲った点は I/O 機器と記憶モジュールの経路を決定する．1つのモジュール線路へのアクセスに対する競合は，優先順位機構により解決できる．クロスバ・スイッチを使うことによって必要な駆動回路の数を減少させることができるが，転送経路を選択するのに必要なスイッチング論理回路は，おそらく図3.9と同じくらいの大きさになるだろう．クロスバ・

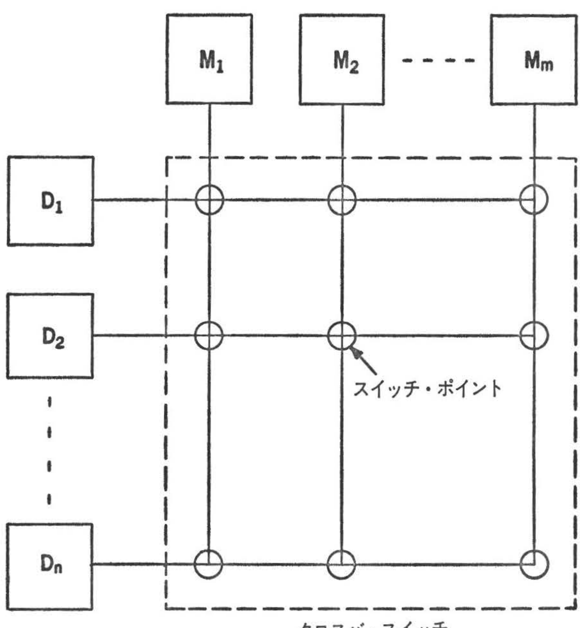


図 3.10　n 個の機器が m 個の記憶モジュールとコミュニケーションするためのクロスバ・スイッチ

スイッチは直接接続方式に比べ，1つのユニット内にスイッチング論理回路を置くことでシステムの信頼性がクロスバ・スイッチの信頼性に大きく依存するという欠点を持っている．

分散接続　信頼性を高めるために，クロスバ・スイッチの論理回路を記憶モジュール内に分散させることが考えられる．その結果の構成は図 3.11 に示されているが，読者はおそらく気付いたと思うが，それは 図 3.1 の構成と同じである．実際のシステムでは，nはおそらくmよりも大きくなるだろう．これはすべての機器が同時に記憶装置をアクセスすることはないことを示している．機器の転送速度が機器によって異なることから，記憶装置へのアクセス経路を共用できる．

共用接続　ある機器は毎秒当たり数情報単位の非常に低速度で転送し，他の機器は毎秒当たり数10万から数100万の情報単位といった非常に高速で転送する．それゆえに低速の機器同志はアクセス経路を共用できる，その方式が図 3.12 に示されている．この方式は時分割（多重化）されたバスを使う．

転送経路を多重化できる機器の接続台数の限界は，その経路のバンド幅と機器の転送速度で決まる．機器の最大転送速度は経路のバンド幅よりも小さくな

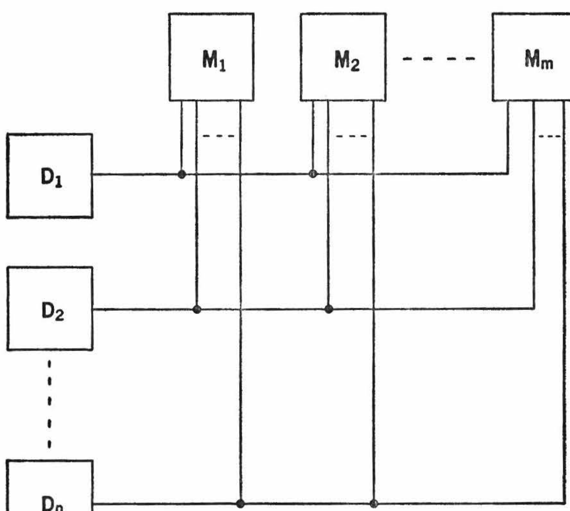


図 3.11　n個の機器とm個の記憶モジュールを接続するための記憶モジュールの中に分散されたコミュニケーション・スイッチング

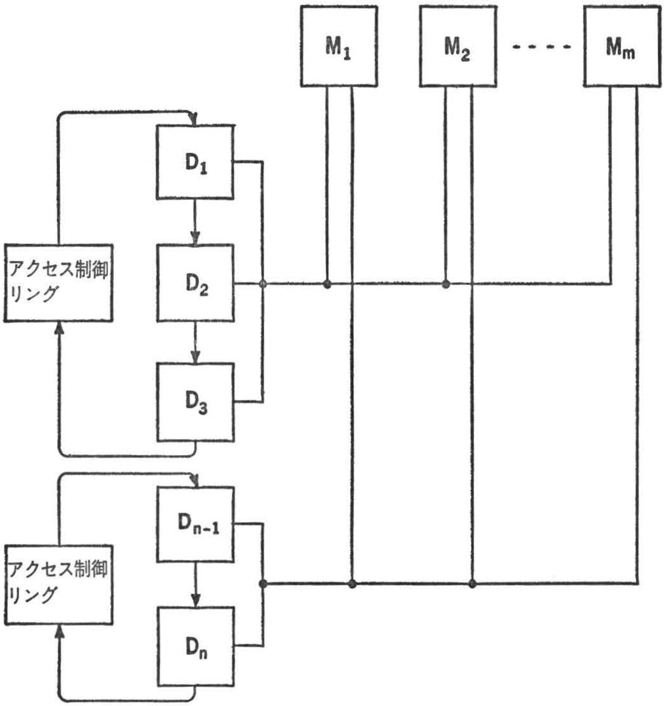


図 3.12 アクセス経路を共用した n 個の機器と m 個の記憶モジュール間のコミュニケーション

ければならない．同時に転送可能なすべての機器の最大転送速度は，記憶バスと記憶ボックスのバンド幅の関数である．この関数は交通量を観察して得た統計的な関数である．転送のピーク時にバスや記憶ボックスの転送速度を越えると，**飽和** (saturation) が生じる．たとえば，すべての記憶バスが飽和以下で動作していたとしても，すべてのバスが同一の記憶モジュールをアクセスしようとすれば，システムは飽和してしまう．設計目標は常にシステムを飽和に近い状態で動作させることである．情報をバッファリング（一時的な記憶）をしないと，飽和状態で情報が失われることがある．機器に必要なバッファリングの量は，システムの統計的特性(記憶モジュールやバスに必要な転送速度の分布)や機器の転送速度，および機器で失う情報の価格あるいは可能性に依存する．

コミュニケーション・システム設計者にとって，目標の情報転送速度を得るために，アクセス経路の数やアクセス経路のバンド幅，バッファリング，および

それらの価格の間に多くのつり合い問題がある．本節での概念は次節で述べる記憶バス構成や I/O プロセッサに接続されるその他の I/O バスに適用する．

3.3.2 アクセス経路の多重化

概 要 共用する転送経路へのアクセスは，通常，その転送経路の接続されているすべての機器を直列的に通過している制御線によって支配されている（図 3.12 参照）．最も簡単な機構はおそらく**制御リング**（control ring）内の位置により，機器の優先順位を割当てるものであろう．転送が完了したとき，制御はアクセスを要求している機器のうち最も高い優先順位のものに渡される．この方式では機器は最悪の転送条件を基に接続される．その機構を使うとしても，転送経路をその最大転送速度で利用することはできない．機器が可変の優先順位でその転送経路にアクセスを要求できるならば，このことは問題にならない．一例として，XDS-940 の場合を考えてみよう．

XDS-940 の共用経路 読者は 3.2.5 節で述べたように，DACC はそのアセンブリ・レジスタがいっぱいになった時に可変の優先順位で主記憶にアクセスを要求できることを思い出してほしい．直接的に主記憶をアクセスする DACC や機器は 図 3.13 で示すように第2記憶バスに接続されている．940 では機器は第2バスに直列的に接続されている．機器の接続されている位置が優

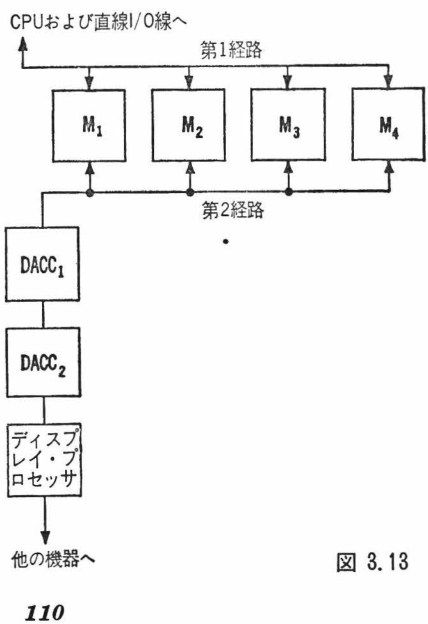


図 3.13 XDS-940 の第2バス構成

先順位を決めるただ1つの要因であるならば，たとえ $DACC_1$ が低い優先順位でアクセスを要求し，$DACC_2$ が高い優先順位でアクセスを要求しても，$DACC_1$ はバスを使うことができる．そのために，第2経路上の機器は，AはBより高く，BはCより高い優先順位でそのバスにアクセスを要求するように少し論理回路が変えられた．最も高い優先順位でアクセスを要求している機器がバスの制御を得る．2台以上の機器が同じ優先順位でアクセスを要求した時は，それらを区別するために，機器が接続されている物理的な位置が用いられる．記憶ボックスとアクセスするための第1と第2経路間の優先順位は次のように決められている．第2経路にある機器がAの優先順位で要求を出すと，第1経路上の機器の要求よりも高い優先順位を持ち，第2経路のBとCの要求は第1経路よりも低い優先順位を持つ．このような可変優先順位機構は，同期式，非同期式システムのどちらでも適用できる．

マルチプレクサ・プロセッサ　これまでは各々の機器が自分自身の記憶アドレス回路，アセンブリ，転送計数論理回路を持っていると仮定してきた．すべての機器を同時に動作させる必要はないことから余分な価格を節約できる．そのために，1セットのアドレス回路，アセンブリ回路，計数論理回路を用意して，これに何台分かの機器を接続する．これを **I/O プロセッサ**という．実際にどの機器に転送を行なわせるかを選択する方法も用意する．この I/O プロセッサに接続されている機器へのアクセス要求は，ソフトウェア・システムによって待ち行列に入れられる．機器が割込み信号システムを通してサービスを要求する様子は4.2節で論じる．

非常に速度が遅い機器が多量の情報を転送する間中，I/O プロセッサを拘束する必要があるならば，それは非能率的な I/O プロセッサの使用方法である．I/O プロセッサをより効率的に使用するために，1台の機器に1単位の情報を転送し，ソフトウェアでこの転送と記憶-転送アドレスを記憶し，他の機器の転送のために切り換えることができる．最初の機器が次の転送の準備ができるとに信号を送って，現在の転送を中止させ，その機器の転送を制御するプログラムに飛び越す．このような方法はしばしば使われるが，CPU の計算の仕事を中断しなければならない．

CPU を使わずに1セットのアドレス，アセンブリ，転送計数論理回路を共用させるために，**マルチプレクサ・プロセッサ** (multiplexor processor) と呼ばれる特別の I/O プロセッサが使われる．このハードウェアは論理的には複数台の I/O プロセッサに見える．各々の論理 I/O プロセッサは固定の記憶領域を持ち，そこには次の転送の記憶アドレス，部分的にアセンブルされた

語，および転送語数が格納される．マルチプレクサ・プロセッサに接続される線路の各々に1セットの記憶領域が付随する．マルチプレクサ・プロセッサは現在で最優先順位を持った機器をサービスする．入力動作では転送アドレス，部分的にアセンブルされた語，転送語数を取出して，機器のサービスを行なう．機器からの単位情報はアセンブルされる．1語が完全にアセンブルされると，マルチプレクサ・プロセッサと主記憶間で転送が行なわれる．そして，記憶アドレスは1加えられ，転送語数は1減らされ，それらはアセンブリ・レジスタの内容とともに対応した固定記憶領域に格納される．出力動作ではマルチプレクサ・プロセッサは，転送アドレス，部分的に分解された語，転送語数を取出して，その機器のサービスを行なう．単位情報がディスアセンブリ・レジスタから取出されて機器に送られる．もしディスアセンブリ・レジスタが空であれば，記憶装置内の転送アドレスから1語が取出されて，ディスアセンブリ・レジスタに入れられる．転送アドレスは1加えられ，転送語数は1減らされ，それらの語はディスアセンブリ・レジスタの内容とともに，対応した固定記憶領域に格納される．最優先順位を持った機器が次に同様な方法でサービスされる．

Multics の汎用 I/O 制御装置　一定の時間内にブロック転送を行なう高速I/O 機器のサービスを行なうマルチプレクサ・プロセッサや I/O プロセッサは単一の記憶バスを共用することができる．これらは図3.13に示したように，1本のバス上に機器別に設置されることもあるし，また Multics システム(図3.4)では1台の汎用 I/O プロセッサ(generalized I/O processor)(GIOC)に統合されている．GIOC はさらに詳しく図3.14に示す．GIOC は Ossana 他によって述べられているように[4]，共通制御，アダプタ制御，およびアダプタ・チャネルといったモジュール化された機能ブロック(modular functional building block)で構成されている．アダプタ・チャネルは I/O 機器の適切な動作終結を行ない，情報のアセンブリやデータのバッファリングを行なう．アダプタ制御は直接および間接の2種がある．**直接アダプタ制御**(direct adapter control)は，記憶転送を直接的に制御するアドレス・レジスタや転送語数レジスタを持つ．**間接アダプタ制御**(indirect adapter control)は最小限度の制御回路しかもたず，低速および中速機器用である．アドレスと転送語数情報は記憶装置内に格納され，各転送が行なわれる時にアクセスされねばならない．間接アダプタはマルチプレクサ・プロセッサに相等し，直接アダプタはブロック転送を行なう機器を1台接続可能な I/O プロセッサに相等する．

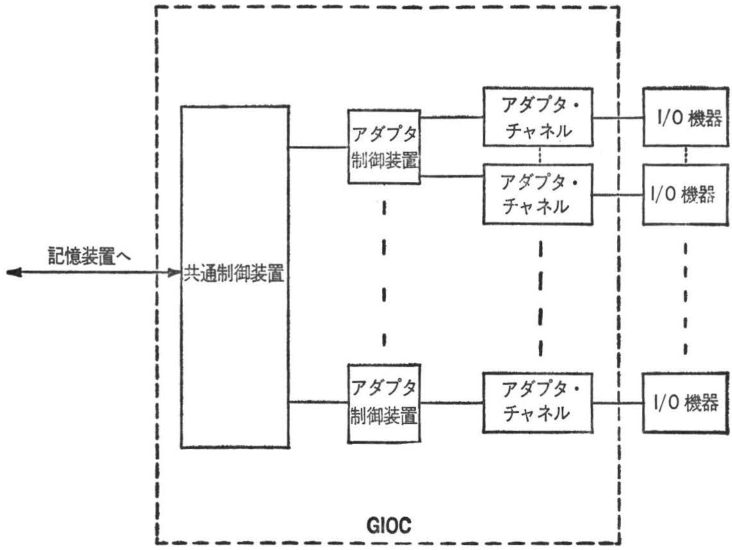


図 3.14 Multics の汎用 I/O 制御装置

共通制御（common control）は，各アダプタ制御に割当てられた固定の優先順位情報を使って，プログラムからアダプタ・チャネルへのデータまたはコマンド情報の転送を行なうために，どのアダプタ制御に記憶データ・バスを与えるかを決める．また，間接アダプタ制御に対するアセンブリや転送語数の更新を行なう．記憶装置への転送が終了した時に，共通制御は転送を要求しているすべてのアダプタ制御を調べて，最も優先順位の高いものを決定する．優先順位の割付けやアダプタ・チャネル内でのバッファリングの方法は，何台もの高い優先順位をもった高速の機器が同時に動作しているような最悪の状態でも情報が失われないことを保証しなければならない．共通制御は間接アダプタ制御の動作を保留し，直接アダプタ制御のための転送を先に行なうことができる．その後，間接アダプタ制御の転送にもどる．間接アダプタは各転送に対し，数記憶サイクルが必要である．このような保留が不可能であれば，直接アダプタ回路に接続された高速 I/O 機器からのデータは失われてしまうこともある．

I/O に対する多重プロセッサの使用　3.4.4 節で遠隔端末機器とのコミュニケーションを制御するために，個別の小型計算機を使うことについて述べた．その概念はその他の I/O を同じように扱うことへ拡張できる．そのよう

な方式はSCC-6700 で採用されている[2]. その考えを 図 3.15 に簡略化した形で示す. すべての I/O プロセッサは同一であるが, 機能を示すために別々の名称が付けられている. これらのプロセッサは CPU の命令セットのサブセットを持っており, 通常の I/O 命令を実行できる. 各 I/O プロセッサは自分自身の記憶装置を持ち, すべてのプロセッサにはシステムの主記憶から I/O プロセッサの記憶装置へデータのブロックを転送する命令がある. 各々の I/O プロセッサはそれに接続された I/O 機器を扱うために, 割付けプログラムと制御プログラムを含んでいる. 補助記憶制御プロセッサは, 記憶装置をドラムやディスクに拡張して扱うためのシステム主記憶に対する特別な高速チャネル

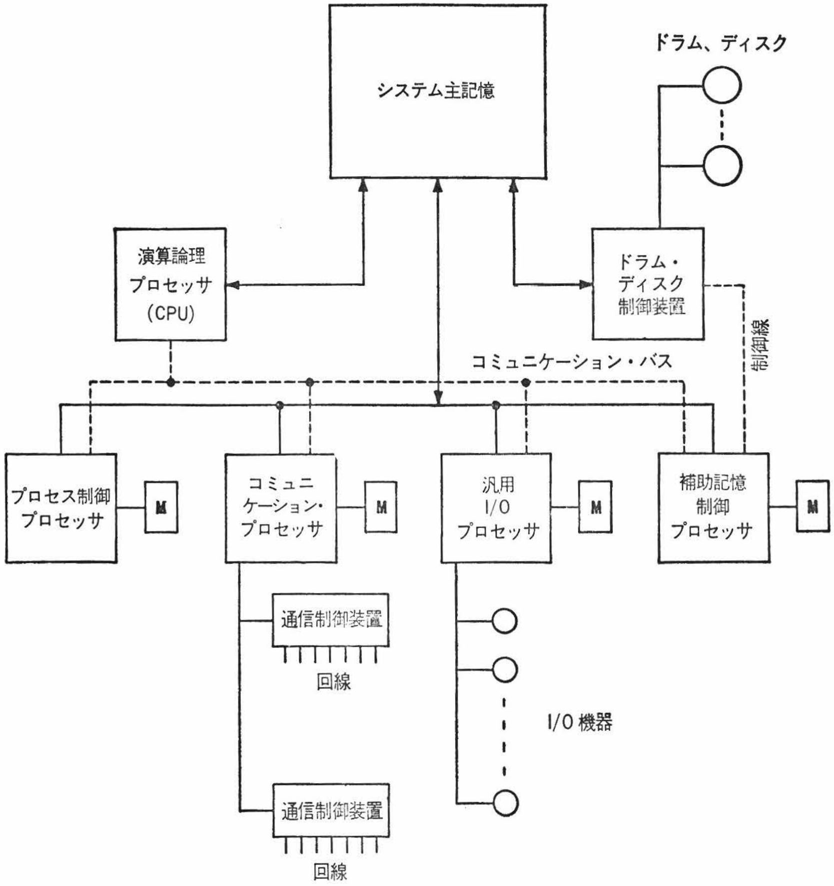


図 3.15　多重プロセッサ I/O 構成

を制御する．プロセス制御プロセッサは，プロセスのスケジューリング機能を取り扱う．各々の I/O プロセッサは，拡張命令セットを経済的に実現するためマイクロプログラム化されている (4.3 節参照)．システム中のすべてのプロセッサは共通の命令サブセットを持っているので，おそらく単一のプロセッサで開発するのと同じ位の容易さでシステムを開発できるだろうし，特定部分のコードは I/O プロセッサに移すことが可能である．

XDS-940 を開発し，SCC-6700 の開発に関係した Calfornia 大学の研究者達はタイムシェアリング・システムを作るために，Berkeley Computer Corporation (BCC) を設立した．以下の記述は，M. W. Pirtle との会話を基にしている．BCC システムには，SCC-6700 のように多数の独立プロセッサがある．しかし，BCC システムでは各々のプロセッサが共通の中央記憶装置に設置されていることが異なっている．各プロセッサは規定されたタスクを持っている．これらのプロセッサのうち 2 台は同一のもので，ユーザのプロセスを実行する．それらは直接的に I/O を実行する能力はもたない．その他のプロセッサはリソース管理を行なうものと考えられる．リソース管理プロセッサの数はシステムの利用方法によって変わる．

通常，3 台のリソース管理プロセッサがあり，そのうちの 1 台は主記憶と補助記憶の管理を，別の 1 台はユーザ・プロセスを実行する 2 台のプロセッサの管理を，そして残りの 1 台はコミュニケーション用の装置と遠隔端末を管理する．また，別のプロセッサがテープ装置やカード機器，プリンタのような主要でない I/O 機器を管理するために必要になるかもしれない．それらのリソース管理プロセッサは，それぞれに割当てられているタスクを実行するように個別にプログラムされている．それらはまた機能を遂行する助けとなるように，マイクロプログラミング (4.3 節参照) といった特別のハードウェア機能を利用している．プロセッサ間で直接的にコミュニケーションする能力はほとんどない．プロセッサ間の大部分の相互コミュニケーションや制御は，共通主記憶内のテーブルを通して行なわれる．BCC システム内の相互コミュニケーションについては，さらに 4.2.4 節で述べる．

3.3.3 ま と め

補助記憶や I/O 機器と主記憶を接続する方法のいくつかを述べてきたが，その中で転送経路，アドレッシング，アセンブリ，および計数論理回路を共用して価格を最小にするために，設計者はいかに機器の転送速度に関する情報を利用することができるかを示した．共用方式における I/O 機器の優先順位を

決定する問題についても述べた．可変優先順位方式で転送経路の融通性と利用率を上げることができることを示唆した．またI/Oを制御するために多重プロセッサを使うことを示した．この方式はシステムの性能を改善する上で多くの利点を与えるように思われる．

I/O 機器へコマンドを送り，そして I/O 機器から状態情報を受け取るための I/O プロセッサおよびシステム上の規約の詳細設計については触れなかった．このことは完全な設計を行なう際には考えなければならない重要な課題である．XDS-940 などのシステムでは，コマンドおよび状態情報はすべての機器に並列に接続された別のバスで転送される．各々の機器はアドレスとして使われるための機器番号を持っている．Multics システムなどでは，コマンドおよび状態情報はデータ転送に使われる線路でもって主記憶から I/O プロセッサを通って機器に直接的に送られる．転送を準備する命令でデータ転送であるのかコマンド転送であるのかが示される．I/O 機器が状態情報を送る時は，特定の信号が出される．I/O プロセッサの設計の詳細は中央処理装置の設計と同様に，本書の範囲外である．

3.4 遠隔端末に関するコミュニケーション

3.4.1 概　　要

本節では遠隔端末とコミュニケーションする概念やユーザとシステム間のインタフェース，計算機と通信回線との間のインタフェースに必要な機器について考えてみよう．コミュニケーションは種々の特殊な分野をもった複雑な課題である．本節ではシステム・プログラマや計算機ハードウェアの設計者が慣れておくべき基本的な概念についてのみ考えることにする．

機器と通信回線の借用料に対する通信価格はタイムシェアリング・システムの価格の中の主要なものの1つである．それゆえに，この課題についてシステム設計者は注意を払わなければならない．

3.4.2 伝送の概念

伝送回線の種類　伝送チャネルまたは伝送回路とも呼ばれる伝送回線は，片方向（simplex）回線，半2重（half duplex）回線，および全2重（full duplex）回線に分類される．片方向回線は一方向のみに情報を転送できる．半2重回線はどちらの方向にも転送できるが，同時には一方向のみである．全

2重回線は同時に両方向に転送が可能である．

転送速度は**ボー**（baud）で表わされ，1ボーは通常1秒間当たり1ビットの転送速度である．電信電話会社は転送速度が110ボーあたりから数10万ボーの範囲で，ビット直列あるいは字直列といった多様な情報転送サービスを行なっている．通信回線の等級は情報を転送できる速度で表わされる．たとえば，通常のダイアル電話回線は**音声回線**（voice-grade line）と呼ばれ，約2000ボーまでの速度で転送できる．要求される転送速度によって，もちろん料金やインタフェース機器は異なる．2000ボー以上の転送には，**交換網**（switched network）とも呼ばれるダイヤル電話回線は使えず，その代わりに専用通信回線を使わなければならない．

変　調　　計算機の機器は図 3.16 に示すように，通常直流2進信号を作り出し，それを利用している．端末機が計算機から，たとえば1/2マイル以内というように近い場合は，これらの直流信号はおそらく単に線を張るだけで直接的に伝送できるだろう．もし端末機が非常に遠く離れている場合は，単に線を結ぶだけでは仮りに不当でないとしても実用的ではない．経済的でかつ正当なただ1つの方法は，電信電話会社が準備した通信回線を利用することである．電信電話会社は一般に直流信号を伝送せずに，**変調**（modulation）という技術を用いている．これはより効率的に伝送でき，雑音による誤まりを起こしにくい他の電気的な形に直流情報を変換する技術である．変調の一般的な形式は，**周波数変調**（frequency modulation）である．この種の変調では，元の信号の振幅を一定振幅をもった周波数の偏差に変換する．計算機のコミュニケーションに一般的に使われる機構は非常に単純である．各通信回線の末端で，計算機あるいは端末機器は**モデム**（modem）（変復調器），あるいは**データ・セット**（data set）と呼ばれる回路に接続される．それは2つの直流レベルを2つの周波数に一端で変換し，他方で2つの周波数を直流レベルに戻すものである．この過程は図 3.17 に示されている．たとえば，“1”ビット（真値）は2000ヘルツの周波数で表わされる，“0”ビット（偽値）は1000ヘルツの周波数で表わされる．これらの周波数では，変換というものは2種の音色を持つと考えられる．先に示した様に，伝送はビット毎に直列に行なわれる．いろいろな一

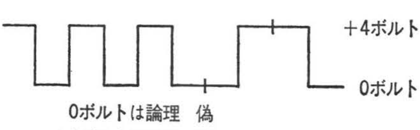


図 3.16　ディジタル装置における論理信号

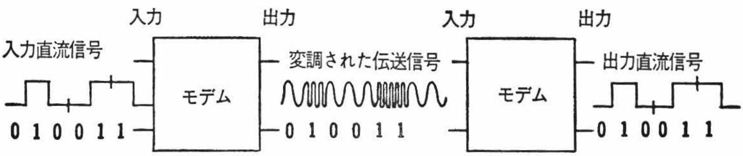


図 3.17 ディジタル伝送のための周波数変調の利用

対の周波数を使って，1本の通信回線で数ビット並列，通常1字送ることができる．このようにして，1本の通信回線は数個の伝送チャネルに分割されている．

コード　字情報は計算機により処理されるためにも，転送するためにも，一連のビットにコード化されなねばならない．多くのコードが使われているが，1つのコード形式，ASCII (American Standard Code for Information Interchange)，がますます使われてきている．128字を7ビットの2進情報単位（ビット）に組み立て，ある種の機器では1ビットの付加ビットが誤まり検出に使われる．このようにして組立てられた字は7ビットあるいは8ビットである．われわれの例では，8ビットの場合を考えることにする．

同期式伝送と非同期式伝送　データ伝送は**同期式**あるいは**非同期式**にできる．非同期式伝送では，字と字の間は不規則な間隔で送られる．字コードに先だって，**スタート・ビット**と呼ばれる1ビットが出され，それは受信機器に対し，次に続く8ビットの字コードをデコードするように合図する．字コードの後に**ストップ・ビット**が続く．これは受信機器に受信を停止させ，次のスタート・ビットを待ち受けるために機器をリセットする．テレタイプに対して，一般に通信回線は字を待っている間“1”の状態にある．スタート・ビットは“0”ビットである．ストップ信号は少なくともリセットする時間を与えるために2個の“1”ビットの長さになっている．したがって，各字毎に11ビット・コードが送られる．上述の概念を図 3.18 に示す．受信側の機器はスタート

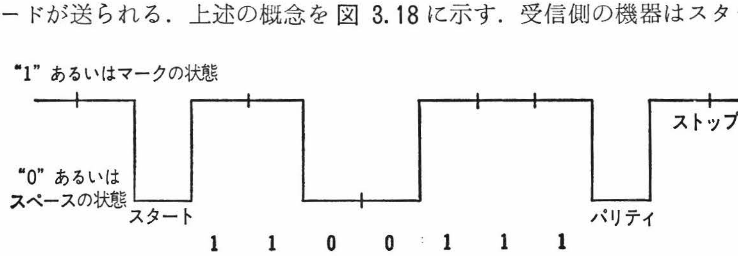


図 3.18 非同期伝送のためのビット列

信号を受信すると，入力される字をデコードするためのタイミング機構をスタートさせる．受信機器は図 3.18 で示すような1ビット時間の何倍かに等しいある時間の間に，一定の電圧で表わされる“1”あるいは“0”の列をデコードできなければならない．たとえば，テレタイプのビット時間は10ミリ秒である．

非同期式伝送は，オペレータが不規則な間隔で字を送るテレタイプやその他の端末機器で使われる．また，送信や受信機器でタイミング発生器の同期の問題を避けたい低速または中速の機器を使って，連続的に伝送を行なう際にも使われる．

同期式伝送は，2台の機械が互いに連続的に伝送を行なう場所や，高速のデータ転送が必要な所で使われる．非同期式では有効なコード情報に含まれない3ビットの同期をとるための情報が必要で，各字について特別な3ビット時間，余分に時間がかかる．同期式伝送では送信と受信機器は正確な同期をとるタイミング発生器を持ち，そのためにスタートとストップ・ビットは必要としない．受信機器は8ビット時間毎に新しい字をデコードする．同期機器は同期を保つために特別に複雑性と注意深さが必要なために高価である．もし受信機器が1ビット時間誤まっても，8ビット時間毎に1字を引出す．しかしながら，誤まり検出ビットを使って，この状況を検出することができる．2台のタイミング発生器は各メッセージの始めに特別な字コードを送ることにより同期をとる．

3.4.3 回線を共用する技術

収集装置　通信回線，特に長距離の高速回線の借料は高いので，このリソースを何台かの端末機で共用するための多くの技術が開発されている．たとえば，商用のタイムシェアリング・サービスは計算機をシカゴに置いて，ミネアポリスやニューヨークやその他の都市にいる顧客を待ちたいのが普通である．これらの客はおそらく長距離料金を払いたくないだろうし，そのためにタイムシェアリング会社は各々の市場と中央計算機の間で1本の通信回線を借り，各市場範囲に**収集装置**（concentrator）を置くようにする．客は地方番号で計算機にダイアルし，収集装置に接続される．客に関する限り，計算機と直接に接続されていることになる．その概念を図 3.19 に示す．

通常，収集装置は小型計算機であり，その領域のユーザから低速の非同期通信回線で送られてくるメッセージを集め，非同期的あるいは同期的に，そのメッセージを計算機に高速専用回線を通して伝送する．メッセージは収集装置に

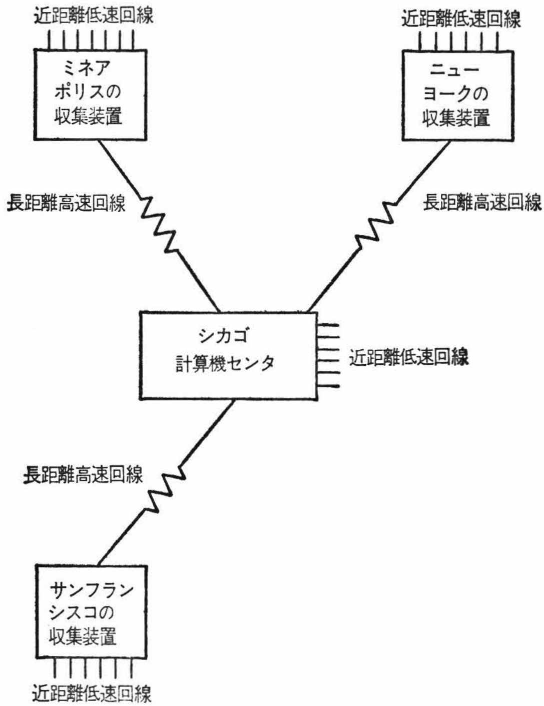


図 3.19　多重高速回線に収集装置を用いたコミュニケーション

よって認識される復帰コードのような特別な1字または数個の字で区分される．収集装置がユーザからのメッセージの終了を示す字を受取ると，ユーザを識別する情報を先頭につけて，そのユーザのメッセージを高速回線の出所として使われるバッファに入れる．

同様に，中央計算機からのメッセージはそのメッセージの先頭にある識別子の収集装置に送られる．収集装置は各ユーザのバッファにメッセージを分類し，低速でユーザにそれを転送する．この技術で幾人かのユーザが高価なリソースを共用できる．これは，タイムシェアリング・システムの設計に関する考え方に一致している．

マルチドロップとポーリングの技術　1本の回線上に何台もの収集装置をもたせたり，何台もの端末機を接続させたいことがある．これは端末機の情報を送るための**マルチドロップ**（multidrop）といわれる技術や端末機からの情報を受取るための**ポーリング**（polling）という技術を用いて実現できる．マル

チドロップではメッセージの先頭に機器アドレスが先行し，メッセージ終了符号が後に続く．メッセージが回線にのせられると，すべての機器はそのメッセージが持っているアドレスをデコードする．そしてそのアドレスを持った機器だけが自分自身で回線に接続する．メッセージ終了符号を受取ると，機器は自分自身を回線から切り離す．通信回線網はいくつかのアドレスをメッセージの先頭につけることを許したり，あるいは，数台またはすべての機器がメッセージを受け取るべきことを示す**ブロードキャスト・コード**（broadcast code）を持つことを許す．上述の概念を 図 3.20 に示す．

計算機へ伝送を行なう際は一時に1台の機器だけが伝送を行なうことができ，回線は計算機の制御下で機器をポーリングするようにできている．**ポーリング・メッセージ**が回線にのせられ，それは“端末Xは何か転送を行なうか？”という意味を持つ．端末Xは“イエス”あるいは“ノー”に対応するコードを送って答える．もし，答が“ノー”であれば次のポーリング・メッセージが送られる．計算機から端末機までの距離が長い時は，ポーリング・メッセージが端末機に送られてから応答メッセージを受取るまでの時間は非常に長くなる．ポーリング方式で若干変わった方式ではメッセージが計算機に送られるか，最後の端末が“ノー”を計算機に戻すまで，“ノー”の応答を次の端末に次々に送っていく．この場合，その端末は適切に識別されるために，そのメッセージの前に機器アドレスを付けなければならない．

中央計算機は通常，ポーリングの順序を含んだ**ポーリング・リスト**（polling list）によってポーリングを行なう．リストの随所に端末のアドレスを入れることにより，種々の端末をポーリングする頻度を変えることができる．これはデータの速度が一定でない機器を統一した方法で扱うための技術として一般に用いられる．ポーリングは特別な端末からデータを集めるタイムシェアリング・システムに広く用いられるようになっている．また，CRT 端末システムを取扱うためにもよく使われている．この場合，多重制御装置のまわりに個々の

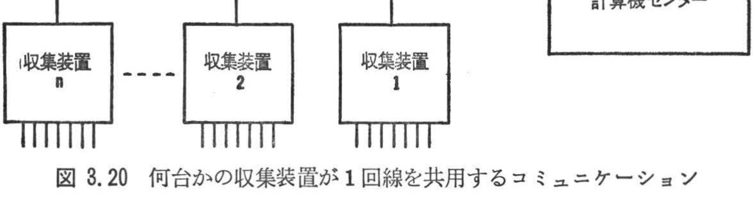


図 3.20 何台かの収集装置が1回線を共用するコミュニケーション

コンソールを置いて独立した装置として扱うよりもずっと経済的に構成できる．本節で紹介した概念は，計算機の I/O システムを設計する時にも使われる．たとえば，XDS-940 の直接 I/O 線はマルチドロップ技術を使っている．

3.4.4 計算機インタフェース

回線バッファ　計算機のインタフェースでは，直列的に入ってくるビット列は字を表わす並列の8ビットに変換し，バッファ・レジスタ内に格納しなければならない．この機能を行なうために，非同期伝送におけるスタートとストップ・ビットを取り除く必要がある．この時に誤まり検出ビットが転送のチェックに使われる．出力においても同様に字を表わす並列ビットを直列形に変換し，スタート／ストップ・ビットを付加する．また，計算機内で字が誤まり検出ビットを持っていない時は，誤まり検出ビットを付加する必要がある．これらの機能は一般に**回線バッファ**（line buffer）と呼ばれる機器によって実行される．その概念を図 3.21 に示す．回線バッファはデータ・セットに対して接続，切断を行なう制御信号を送受し，適当な同期をとるための別の機能も実行する．また，回線バッファはモデムから，たとえばそれらが回線にまだ接続されているか否かを示す状態情報も受け取る．この状態情報は計算機で調べることができ，あるいは回線の状態が変化した時に信号を発生するために使うことができる．これらの機能は回線の自動応答を行なうために重要であり，また事故あるいは誤まりで切断した場合のシステム障害を避けるためにも重要である．このような切断が生じた時には回線を解放し，ユーザを保護するためにその回線を使っているユーザを自動的にログ・アウトすることが必要である．各回線に対して1個の回線バッファが設置される．

通信制御装置　次に，回線バッファは主記憶とコミュニケーションしなければならない．このコミュニケーションは回線バッファを通常，**通信制御装置**と呼ばれる装置に接続して取扱われる．通信制御装置を GE-645 のようにマルチプレクサ・チャネル（間接アダプタ制御装置）に接続したり，XDS-940 のように直接 I/O 線に接続したり，あるいは GE-265 のように中央計算機とつながった個別の小型計算機に接続することもできる．その小型計算機は回線バッファ機能や通信制御装置の機能をプログラムで直接扱えるように設計されている[5]．

XDS-940 の通信制御装置は次のように動作する．まず入力について考える．回線バッファが1字を受け取ると，入力フラグをたてる．通信制御装置は常に回線バッファを順次調べており，入力フラグをたてている回線バッファを発見

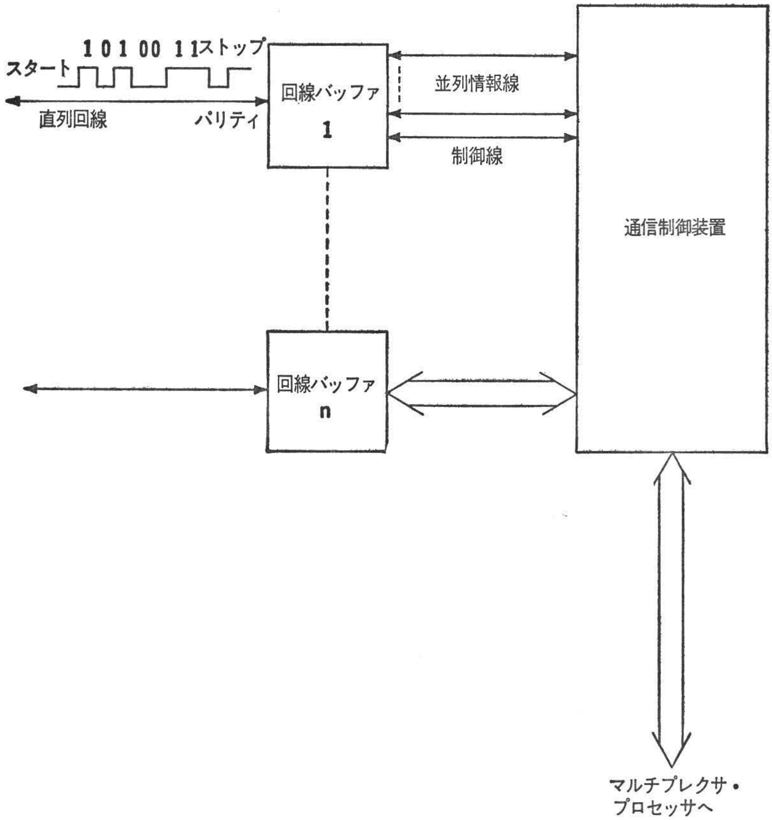


図 3.21 計算機と通信回線のインタフェース

した時は調べを中止して，字コードと回線のアドレスを含む1語を組立てる．この語は**データ・オーバーラン・ビット** (data-overrun bit) を含んでおり，そのビットは回線バッファがサービスされる前に2字が到着した時にセットされる．組立て終了後，制御装置は割込み信号を発生し，XDS-940 は組立てられた語を主記憶内に入れるための直接入力コマンドを出す．もし制御装置がマルチプレクサ・チャネルに接続されているようなときは，サービス要求を示して，マルチプレクサ・チャネルは語を主記憶内に格納することになる．

出力の場合は，もう少し複雑な手順が必要である．回線バッファが1字の出力を完了した時，出力完了フラグを立てる．通信制御装置は出力完了フラグを発見すると走査を中止し，XDS-940 に割込みをかける．XDS-940 は回線のア

ドレスを取出し，さらに字を出力するかどうかを調べる．もしそうであれば，940 は次の字とその回線番号を含む1語を組立て，それを制御装置へ送る．制御装置はその字を適切な回線バッファに送り，走査を続ける．マルチプレクサ・チャネルを使ったシステムでは，サービスの要求は直接に次に転送する字を取出すための記憶参照を作り出す．通信制御装置は自動的にポーリングし，マルチドロップ機能を実行する装置としてみることができる．

XDS-940 のように，直接 I/O 線を使うことは各字を処理するのに約 300 マイクロ秒の CPU 計算を必要とする．この時間は限られた数（32以下）の低速回線が使われている時にはあまり問題にならないが，いくつかの高速回線や多数の低速回線が使われる時には深刻な問題になる．それゆえ，マルチプレクサ・チャネルや小型計算機に通信制御装置を接続するのはおそらく良い方法であろう．

タイムシェアリング・システムは 110 から数1000ボーにわたる多様な転送を行う端末を扱うように次第に要求されている．新しい機器はハードウェアやソフトウェアの中刻部分の変更なしに簡単に接続できることが重要である．これらの目標は先に述べたような各回線を順次走査する単純な通信制御装置や可変優先順位が通常必要な通信制御装置では簡単には達成できない．そのような機器はあるサービス要求を終了した後に，サービス要求を出しているすべての回線バッファを同時に走査し，その中から最も優先順位の高いものを次に処理する．

個別の小型計算機に通信制御装置を接続することにより，中央計算機の動作に関係なく，編集やメッセージの組立て，ファイルの作成等多くの処理を行なうことができる．この有用な機能については，さらに 6.2.5 節で述べる．

3.4.5　端末インタフェース

端末は，ユーザとシステム間のインタフェース部分である．人と機械の能力の調和がとられるのはこの部分である．メーカは多くの有用な端末を生産しているが，それらはタイプライタ，CRT，そして特殊用途端末に分類できる．読者にソフトウェア設計者の広範囲の端末を取扱う場合に直面する複雑性について理解してもらうために，端末のいくつかの特性について示すことにする．

タイプライタ端末は種類により 110 ボーから 400 ボーあたりまでの速度で送受信を行なう．これらの端末のあるものは半2重モードで動作でき，その場合はキーボードは直接的に印字機構に接続している．そして各々のキーがたたかれると，ただちに印字紙の上に記録される．その他に全2重モードで動作でき

るものがあり，その場合は印字機構は論理的にキーボードとは独立にされている．キーがたたかれると字が計算機に送られ，逆にその字を呼び戻して，その結果を印字させる．全2重モードにおいては，入力と出力の受信を同時にできる．また全2重モードではその字が正しく受け取られたかを有効にチェックできる．もし正しい字が返って来た時には，正しく受け取られたことを知ることができる．あるタイプライタ端末は1字毎に字を計算機に送り，またあるものは1行あるいは1メッセージ毎に送る．あるものは復帰あるいは改行を1字の時間で行ない，またあるものは復帰時間はまちまちのこともある．種々の制御あるいは字コードが使われる．

CRT 端末は，2400 ボーまでの速度で受信できる字形式端末からマイクロ秒以内に線や字を描くことのできる汎用グラフィック端末に至るまである．これら汎用グラフィック端末は記憶バス，I/O プロセッサ，音声級あるいはより高速回線を持った小型計算機に直接接続することができ，それは色々な方法で中央計算機に接続できる．汎用グラフィック端末とそのインタフェースの設計はそれ自体特別の課題である[6]．字形式 CRT 端末は自分自身でバッファ記憶を持っており，一般に半2重モードで動作して情報をブロック単位で転送する．ある最新の端末は全2重モードで動作し，キーボード動作は全2重モードのタイプライタ端末のように実行し，表示のために自分自身の記憶装置への出力は他の CRT 端末のように取扱われる．CRT 端末は計算機に独立した装置にすることもできるし，何台かの端末を1つあるいはそれ以上の制御装置に接続することができる．しかもそれらの制御装置は1本の回線に接続されている．この端末のために種々の字や制御コードが使われている．

特殊端末は先に述べた端末のように多様であり，入力用キーボードあるいは他のコード発生機構，出力用にプリンタあるいは CRT を持っている．

特別に端末インタフェースに必要なものは，ソフトウェアで要求を処理できるように端末のサービス要求の形式を識別する何らかの方法である．先に述べた“転送”と“計算機インタフェース”に関する基本ハードウェアの概念は，幅広い端末にも同様に適応できるが，各端末の形式はソフトウェアに対して色色な問題を起こす．それゆえ，一番最初にシステムと接触する際には，端末の形式を識別することが必須となる．現在のところ，この情報システムに送る標準的な方法はない．テレタイプの呼出しに対する応答ではシステムは端末が転送するのを待っていれば良いが，多くの CRT 端末のような端末では，情報を受け取るためには計算機に選択されねばならない．ポーリングに対する決まったコード順序がないと，困難な問題がもち上がる．システムがいかにして情

報を選ぶのか，あるいは待つかを知るかという問題である．一方法はたとえば30秒待ってから選択するという方法である．識別に対する一般的な方法は各端末の形式毎に異なった番号をダイアルさせることである．

3.4.6　ま　と　め

この節では回送線を介して中央計算機に接続された遠隔端末の情報転送を含む一般的な概念を紹介し，そして，端末の性質や識別の問題について考えてきた．収集装置やマルチドロップ，ポーリングといった多くの考えは I/O システムの他の分野にも適用できる．端末の性質や識別の問題は，もしシステムにユーザの要求に応じて種々の機器を簡単に付けられるようにするには回線のインタフェース機器の設計に十分に注意を払う必要があることを読者に警告している．

参　考　文　献

1　**Pirtle, M. W.:** Intercommunication of Processors and Memory, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 621–633, 1967.

2　"6700 Time-sharing Computer Reference Manual," Scientific Control Corporation, Carrollten, Texas, February, 1969 (additional information was also obtained at a public presentation of the 6700).

3　**Conti, C. J., et al.:** Structural Aspects of the System/360, Model 85, *IBM Systems J.*, vol. 7, no. 1, pp. 2–29, 1968.

4　**Ossana, J. F., et al.:** Communication and Input/Output Switching in a Multiplex Computing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, 1965.

5　**Newport, C. B.:** Small Computers in Data Networks, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 773–775, 1969.

6　**Watson, R. W., et al.:** A Display Processor Design, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 35, 1969.

第 4 章

保護および制御

4.1 システムの保護

4.1.1 概　　要

リソース共用システムにはハードウェアおよびソフトウェアの双方で多くの保護レベルが必要とされる[1,2]．リソース共用システムはここ10年の間運用されてきたにもかかわらずまだ開発段階にあり，保護機構の範囲や実現された種々の保護機構および実現可能な価格は，まだ十分わかっていない．計算機システムの設計は，一般に最も汎用の機構だけをハードウェア化したり，明らかにわかるシステムの欠陥をハードウェアで取り除くことしかしていない．そのようなことから，保護方式の多くがハードウェアよりむしろソフトウェアのルーチンの中に存在している．現在のシステムについて経験が加えられ，そして設計者が種々の保護方式についての価格に関する統計を収集するにつれて，また，その機構がいかに一般化され，互いにどのように関係しているかがわかるにつれて，ハードウェアによる保護が発展するであろう．本節では，現在ハードウェアで存在する保護の方法について検討し，ソフトウェアについて存在する保護機構については，5.4節にまわす．次のような種々の保護が Lampsonによって区別されている[1,2]：

1. ユーザ・プロセスからのシステム保護

   ユーザがシステムの実行を止めたり，システムにとって基本的な情報を壊すような動作ができないようにするのは絶対に必要である．

2. ユーザ相互間の保護

   ユーザに他のユーザのプロセスの実行を妨害する動作を許してはならない．

3. ユーザ自身からの保護
ユーザは実行中のいくつかのプロセス，あるいはいくつかのサブプロセスを含む1つのプロセスを持っており，これらのすべては相互にコミュニケーションしている．ユーザは記憶装置のアクセス，実行時間，あるいは交信の経路の面で，これらのプロセスを互いに保護したい場合がある．
4. システム自身からの保護
タイムシェアリング・システムは常に発展している複雑な実体である．故障したモジュールが起こす損害を制限する保護機構が不可欠である．

ハードウェア設計の目的のために，上に述べた保護に対する要求は主に2つの点に縮小される：

1. 計算機の物理記憶内のある語に対して，アクセスしたり，変更したり制御を移さないようにする保護（記憶保護）．
2. ある命令を実行しないようにする保護（制御保護）．

ハードウェアの機能が与えられると，ソフトウェアの設計者はシステム全体の保護方式を開発するために，これらの機能を色々な方法で利用できる．本節では，現在および計画中の計算機において見られる種々のハードウェアの保護機能について紹介し，後で（5．4節）ソフトウェアの観点から再び保護機能を取扱う．

4.1.2 記憶保護

概　要　記憶保護方式においては，記憶は次のような種々の区別が行なわれる：

1. アクセス不可（inaccessible）
2. 読出し，書込み可能（read write）
3. 読出し専用（read only）
4. 実行専用（execute only）

色々な保護の種類を使った例は，次のようなものがある；多くのプログラムはモニタ・システムと同様にある時間主記憶内に存在する．それゆえ，記憶のある領域からは他の領域をアクセスできないようにする必要がある．ほとんどのシステム・プログラムには，すべての領域に対して読出し書込みができる特権が必要である．コンパイラ，エディタ，デバッグ手段などのサブシステムは，

ライブラリ・ルーチンと同様に多くのユーザによって共用される．そのため，それらを読出し専用にすることによって，適切に保護できる．計算機の事業では，特別なプログラムをユーザに貸す時間は，おそらく重要な業務になるだろう．占有しているプログラミング技法や他の技法を保護するためにユーザがそのコードを読まないようにしたいが，一方ではそのコードを実行できるようにしたい．このような場合には，実行専用という保護が必要となる．

物理アドレス空間の保護　記憶保護は物理アドレス空間，または論理アドレス空間に対して適用することができる．物理アドレス空間を保護するには，物理記憶を固定した大きさのブロックに分割し，各々のブロックについてハードウェア・レジスタとして保護キーを割り当てる．このキーは，対応したブロックに許されるアクセスの種類を示す．この方式における困難な点は，それが各プロセスを区別していないことである．どのプロセスでも書込み可能のキーを持つブロックには書込むことができ，実行可能のキーを持つブロックでは実行することができる．情報を共用するシステムでは，あるコードに対して，ある人には書込み可能の特権を与え，その他の人には読出し専用とする必要がある．そのようなシステムでは，システムが別のプロセスに制御を切換える度にソフトウェア・モニタは保護キーを変えるために記憶装置を調査しなければならない．この方式はシステムのオーバーヘッドを増加させ，結果として扱いにくい設計になりかねない．

この事態を何とかして改善しようとする試みとして，あるシステムではキーーロックの方法を用いている．上で述べたように1組のロックが記憶ブロックに割当てられ，各プロセス毎に状態語としてキーが渡される．もし状態語がそのブロックのロックと指定された論理関係にあれば，特定の階級のアクセスが許される．この方式でも，まだ次に示す方法ほど望ましいものではない．なぜならば，不注意にブロックへ書込むことができるからである．これを監視するには非常に大きなキーが必要である．現存するいくつかのシステムでは4ビットのキーを使っているが，これは16個の組み合わせしか得られない．このようなシステムによる保護機構は，おそらく幅広いソフトウェアの介入が必要となろう．

論理アドレス空間の保護　上に述べた問題を回避して保護をさらにうまく応用できる場所は物理アドレス空間ではなく，むしろ論理アドレス空間を保護することである．論理空間を物理空間にマッピングするために再配置レジスタやベース・レジスタを使ったシステムでは，しばしば**限界レジスタ** (bounds register) で保護をかけている．限界レジスタは記憶領域を2つの部分に分割

する．すなわち，ベース・レジスタと限界レジスタで指定される場所にはさまれた連続領域とその他の部分である．この方式を 図 4.1 に示す．プロセスは前者の領域にアクセスすることはできるが，後者の領域に対するアクセスは拒否される．一般に，各々のベース・レジスタに対して限界レジスタが設けられる．この方式は先に述べた記憶保護方式よりも改善されている．なぜならば，計算機が制御するプロセスが変わるときに保護を容易に変えることができるからである．しかしながら，それにしても制約がある．限界レジスタ方式による保護形式では，保護をすべてかけるか，あるいはかけないかのどちらかになる．タイムシェアリング・システムでは，一般に保護に関してもっと細かな区別が必要である．保護は連続した範囲に限られているので，プログラムを共用する際は少なくとも2つのベース・レジスタ-限界レジスタ対を使う限り，頻繁にそれらを変える必要がある．

最良と思われる保護方式は，ページングやセグメンテーションのハードウェアまたはその両方を持っているシステムに見ることができる．IBM 360/67 は設計に具合の悪い面が目立っているように思われる．このシステムは精密なセグメンテーション・ハードウェアを持っているにもかかわらず，記憶装置の物理的なブロックに保護がかかるようになっている[4]．　ページングあるいはセグメンテーション・ハードウェアによる保護方式では，あるプログラムにタスクを遂行するために必要なだけのプログラムおよびセグメントへのアクセスを容

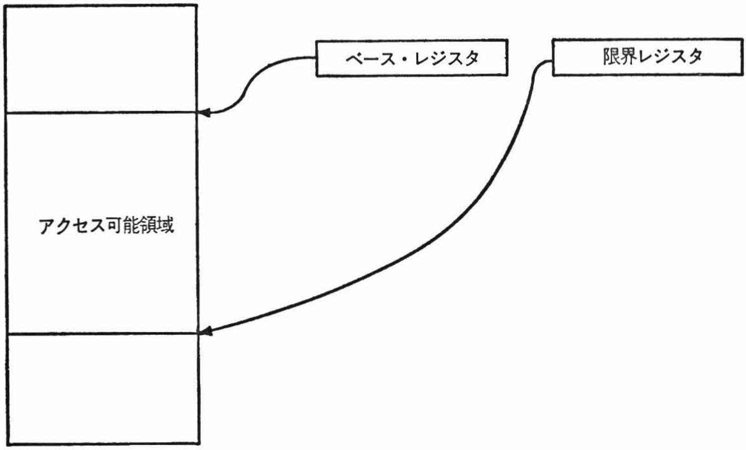


図 4.1　限界レジスタによる論理空間の保護

易に与えるようにでき，しかもそのタスクに必要なアクセスの様式だけを与えることができる．

たとえば XDS-940 では，2.4.2節で述べたマップ・レジスタを利用した簡単な保護方式が行なわれている[3]．各マップ•レジスタは6ビットから成っているが，ブロック・アドレスのためには5ビットだけが使われ，残りの1ビットは保護の種類を定義するために使われていることを思い出してほしい．もし保護ビットが“1”であれば，そのページは読出し専用である．すなわち，そのページから情報を読出すことはできるが，いかなる情報もそこに書込むことはできない．もし読出し専用のページに書込もうとすると，誤り割込みが生じ，適当なシステム・ルーチンに制御が移される．保護ビットが“0”であるときは，そのページがユーザに割当てられたものであれば，ユーザはそのページを読んだり書いたりできる．ユーザに割当てられていないページに対応したマップ・レジスタの保護ビットは“1”で，その他のビットは“0”になっている．このことは，ユーザは実際のブロック0には決して割当られることがないことを意味する．このブロックは常にシステムが使う．他のテーブル内に保持されている情報により，マップ・レジスタに許されるアクセス様式がセットされる．

より一般的な保護システムは GE-645 に使われており，保護はセグメントのレベルで行なわれる[1]．セグメント・ディスクリプタ語に格納されている情報の一部は，そのセグメントに対して許されているアクセス様式である．すなわち，書込み専用，読出し専用，読出し／書込み可能，実行専用，アクセス不可，システム・レベルのプロセスに対してのみアクセス可能，などである．この保護情報は，あるプロセスがそのセグメントに対してアクセスを要求したときに，セグメント・ディスクリプタにセットされる．セグメントはディレクトリを介してファイル・システム中に位置付けられるが，ディレクトリはセグメントが格納されている補助記憶の場所だけでなく，そのセグメントに対してどのユーザにどの様式のアクセスが可能であるかをも示す．この情報によって，セグメント・ディスクリプタ語がセットされる．

GE-645 にはその他の保護レベルがあり，IBM 360/67 にも似た方式が存在する[4,5]．ディスクリプタ・ベース・レジスタおよびセグメント・ディスクリプタの中には，各々定義したセグメントの最大数と最大セグメント長を示す限界フィールドがある．一般アドレスの2つの要素がこれらの限界値と比較され，この限界内になければ記憶アクセス違反になる．連想マップ内のエントリーもまた固有の保護情報を含んでいる．

保護はセグメントに対するアクセス経路ではなく，むしろセグメントに対し

て定義されるので，上記の方式では，潜在的な難しさがある．すなわち，プロセス内のいかなるセグメントも，与えられたセグメントに対してそのディスクリプタで定義されたアクセス権と全く同じものを持つことになる．その難しさは，デバッグ・プログラムを例にとると理解できる．デバッグ・セグメントは，それ自身に対してあるいはデバッグされるプログラムに対して読み書きするアクセスを必要とする．ユーザ・プログラムのセグメントはデバッグ・セグメントに対してアクセスすべきでないが，自分自身に対しては完全なアクセスをする．デバッグ・セグメントとユーザ・プログラムのセグメントの両方が，1つのセグメント・ディスクリプタ・テーブルを持つプロセスを形成する．そこで問題になるのは，どのようなアクセスの権利をデバッグ・セグメントのセグメント・ディスクリプタに与えるべきか，ということである．GE-645 システムでは，この論争をセグメントへのアクセス経路のハードウェアによる保護よりも，むしろ全体をソフトウェアの保護システムにすることで解決している．

Evans と LeClerc[6] によるセグメンテーション方式は，保護接続行列を用いることによりアクセス経路に対して保護を行なっている．GE-645 のようなシステムで同じような結果が得られている．1つのプロセスは，n 個のセグメントを持つことができる．各セグメントは，他のセグメントに対するアクセス経路を n 個持っている（自分自身を参照するものを含む）．このように，各セグメントは n 個の可能性から成るアクセス・ベクトルに割当てられる．あるセグメントは，セグメント・ディスクリプタ・テーブルにおける j 番目のセグメントに対してアクセス・ベクトルの j 番目のエントリーに示されるアクセス様式が与えられる．アクセス・ベクトルは，ディスクリプタ・セグメントとともに記憶装置に格納され，ハードウェア・マップで実現される．

セグメンテーションに対する接続行列の方式は，SCC-6700 に実現されている．アクセスの保護はセグメント間の接続経路について行なわれるが，これはアクセス不可，読出し可能，読出しまたは間接（書込み不可）可能，あるいは，任意のアクセス可能のいずれかである．ハードウェア・マップに割当てられるものとして4個の**セグメント間アクセス・レジスタ**（intersegment access register）（ISAR）があるが，これは4つのセグメントの各々の8つのセグメントに対するアクセスの特権を定義する．ある ISAR に割当てられているセグメントがその ISAR に書かれていないセグメントをアクセスすると，特別なマイクロプロセッサ（2.5.8 節参照）が，ソフトウェア・テーブルに格納されているアクセス情報によって ISAR を更新する．アクセス経路に対して行

う保護の概念を図 4.2 に示す．図 4.2(a) は，10 個のセグメントを持つプロセスにおけるアクセス保護行列である．セグメント n のアクセス・ベクトルは，

セグメント＼セグメント	1	2	3	4	5	6	7	8	9	10
1	3	0	3	2	0	0	1	0	0	0
2	1	2	1	1	0	0	1	0	1	2
3	0	1	3	1	0	1	0	2	0	3
4	1	2	2	3	0	1	0	0	1	2
5	0	1	3	1	3	0	1	2	0	1
6	3	3	3	0	1	2	3	0	3	2
7	0	0	1	0	1	3	3	1	3	0
8	1	1	3	1	2	1	0	3	2	1
9	1	0	3	0	1	0	2	3	3	1
10	0	1	2	3	3	3	1	1	0	3

0 アクセス不可
1 読み出し専用
2 実行専用
3 読み書き実行可能

(*a*) アクセス保護マトリクス

A	B	C				
3	0	0	1	3	1	0
8	1	1	0	3	2	1

フィールドAはアクセス保護マトリクスの行のセグメントを示す。
フィールドBはCフィールドが前半の5つのセグメントかあるいは後半の5つのセグメントに対するアクセス権を含んでいるかを示す。 0＝前半の5つのセグメント
1＝後半の5つのセグメント
•フィールドCは5つのセグメントに対するAフィールドのセグメントのアクセス権を示す。(セグメント 1, 2, 3, 4, 5あるいはセグメント 6, 7, 8, 9, 10)

(*b*) 2個のセグメント間アクセス・レジスタ

A	B	C				
3	1	1	0	2	0	3
8	1	1	0	3	2	1

(*c*) 2個のセグメント間アクセス・レジスタ

図 4.2 セグメント間アクセス経路における保護機構

行列のn行によって与えられる．セグメントnがセグメントmに対してアクセスする権利は，n行m列の要素で与えられる．たとえば，セグメント3はセグメント8に対して実行専用のアクセスを行う．完全なアクセス保護行列はソフトウェアで実現される．アクセス保護行列全体をハードウェアで実現することはおそらく不経済であり，不必要でもある．

図4.2(b)は，2つのハードウェアのセグメント間アクセス・レジスタがどのように実現されるかを示している．この機構は本質的には，SCC-6700のものと同一である．5つのセグメントに対するアクセス権がハードウェアに含まれている．もし，他の5つのセグメントにアクセスが行なわれると，それがセグメント3からであれば，セグメントに対応しているISARはリセットされ，図4.2(c)に示されているように他の5つのセグメントに対するアクセス権が保持される．

この概念はまた，セグメント番号とアクセス権をISARに明白に関連付けることによっても実現できる．この方式は，ISARにないセグメントが参照された場合にすべてのエントリーを置き換える必要はなく，ISARの個々のエントリーを変えるだけで良い．ISAR内のどのエントリーを置き換え別の情報を持たせるかを決めるには，おそらく多少ハードウェアが複雑になるであろう．

GE-645のようなシステムはセグメント間のアクセス経路に対するハードウェアの保護は行なっておらず，5.4節で述べるように，そのような保護をソフトウェアで作り上げている．5.4節では，Multicsで使われている保護システムのための付加ハードウェアについて述べる．

4.1.3 制御保護

概　要　常にではないが，一般にある種の命令はプロセスが実行しようとしても許されないことがある[3-5]．たとえば，プロセスはI/O命令を直接実行しようとしてもできないのが普通である．なぜならば，他のプロセスが特定の機器を使っている場合があるからである．すべてのI/O動作要求は一般にシステムを通るが，それはサービス要求をスケジュールできるようにするためである．さらにユーザのプロセスは計算機を止めたり，システムや他のユーザに妨害を与えるような命令の実行をしないようになっていなければならない．

ユーザおよびシステム・モード　上記の問題を扱うために多くの計算機は2つのモードで動作するように設計されており，一般にこれらは**システム・モード**および**ユーザ・モード**と呼ばれる．システム・モードでは，すべての機械命令が実行できる．ユーザ・モードでは，**特権命令**（privileged instruc-

tion）と呼ばれるある種の命令の実行は禁止される．もし特権命令を実行しようとすると，誤り割込みが生じ，システムに制御が移される．たとえば XDS-940では次の命令は特権命令である．すなわち．すべての I/O 命令，すべての割込みを制御する命令，すべてのコンソール・スイッチや I/O 機器の状態を読取る命令，すべての使われていない操作コード，および停止命令である．

特権命令の実行を必要とする動作を行うために，ユーザ・プロセスはシステムを呼び出す．システムと交信する能力は，ユーザがタイムシェアリング・システム下で実行する独自のシステムを開発する上で大きな融通性を与える．さらに用いられるアドレス方式によって異なるが，2つのモードでアドレスの仕方が異なることがある．たとえば，XDS-940 には2つの記憶マップがあるが，その1つはシステム・ルーチンのためであり，他はユーザ・ルーチンのためのものである．940 には，またシステムがユーザ・マップをも使用できる簡単な方式があるが，これはモード間のコミュニケーションを容易にし，リエントラント・ルーチンを書くのが簡単になる．多くのシステムでは，システム・ルーチンが絶対物理アドレス参照を行う必要がある．このようにすべてのシステム・ルーチンは完全な読出しおよび書込みアクセスを持っている．この方式では，システムはまだソフトウェアやハードウェアの誤りによって傷つきやすいままである．システムがマップを通して記憶装置を参照する場合は読み書きを要求するシステム・ルーチンを一組のページ内に入れることができるし，またその他の一組のページに読出し専用アクセスを要求するシステム・ルーチンを入れることができる．この方式によれば，ハードウェアおよびソフトウェアの誤りをさらに容易に局限することができる．

ユーザ・モードとシステム・モード間の切換え　モード間のコミュニケーションの問題には注意深い考察が必要である．なぜならば，もしモード切換え方式の設計が注意深く考えられたものでない場合には，プログラミングがしにくかったりモード間の呼び出しの正当性を特別に調べることに時間が余分にかかたりする．

最も簡単なモード切換え方式は XDS-940 に実現されているが[3]，そこでは**プログラム化されたオペレータ**（programmed operator）を使っている．プログラム化されたオペレータの概念は，要するに，ユーザに対してサブルーチンの呼び出しを機械命令として論理的に生じさせる方法である．プログラム化されたオペレータを意味する命令語のビットが検出されると，通常操作コードとして解釈されるビットが制御を移すアドレスとして解釈される．XDS-940 には，2種のプログラム化されたオペレータがある．すなわち，(1)プログラム

化されたシステム・オペレータであって，これによりシステム・コード内の記憶場所に制御が移され，モードがシステム・モードに変わる（前のモードは退避される）ものと，(2)プログラム化されたユーザ・オペレータで，これによりユーザ・プロセス内の記憶場所に制御が移されるものである．したがって，システムの助けを求めるためのすべての呼出しは，プログラム化されたシステム・オペレータが行う．システム・モードからユーザ・モードに戻るために，システム・マップよりはむしろユーザ・マップを通してアドレス付けされる飛越し命令を実行する．システム・モードである間，システムはユーザ・マップを用いてユーザ・プロセスの記憶場所をアクセスすることができる．さらにモード間に簡単なインタフェースを与えることにより，プログラム化されたオペレータの概念は，ユーザにむきだしのハードウェアからなる計算機ではなく，もっと能力のある計算機に対してプログラミングしていると思わせる．

Multics システムでは異なったモード切換え方式が使われている．Multics では，共用されるシステム・セグメントは各プロセスに割当てられ，各ユーザのセグメント・ディスクリプタ・テーブルの中にエントリーを持つ．システムへの飛越しは，通常のセグメント間の参照と同じ方法で扱われる．システム・セグメント・ディスクリプタの制御ビットは，そのシステム・セグメントがシステム・モードで動作していることを示す．Multics のモード切換え機構は，システム呼出しがすべてのセグメント間の参照により一様に扱われるという意味で一般的であり，完成されたものである．

記憶領域に関連した制御保護　上述したユーザ－モニタ・モード方式において，モードの級別はプロセスに対応している．モードの級別を記憶領域に対応させるような方式が提案されており，それによって制御保護の問題を記憶保護システムに結びつけている．その結果は，ソフトウェアの設計において融通性を増加させる利点がある．ユーザ－モニタ・モード方式には，ただ2種類の保護があるだけである．すなわち，1つはプロセスがシステムに対して完全なアクセスを持っているが損傷を与える可能性もある場合と，プロセスは制限されたアクセスを持つ場合とである．さらに，ユーザ・モードに類別されているプロセスが，特権命令を実行する必要がある機能を行うためにはシステム・ルーチンを呼出さなければならない．そのような要求を出す権利を判断する責任はシステム・ルーチンにある．この種の検査は各呼出しに対して行なわれなければならないので，時間の浪費となる．

記憶保護の部分として制御保護を用意することは各呼出しに対するこの検査を取り除き，保護を利用する場合の融通性を増す．この検査の問題は保護をセ

グメントのアクセス経路に対してではなく，セグメントに割当てるために起こる問題に大変良く似ている．システム・ルーチンのアクセス権を検査することは，システム・ルーチンへのアクセス経路に対する保護をソフトウェアで実現するものである．ここで述べる方法は，ハードウェアの助けによってアクセス経路に保護を行うものである．

Lampson[2] により提案された制御保護方式は，以下のように動作する．プロセスが要求する特権命令はシステムによって実行専用と印された記憶領域に入れられ，特権命令を実行しても良いという状態情報が付加される．そのプロセスはこれらの特権命令を実行命令でアクセスする．（実行命令はアドレス・フィールドで指定された記憶場所の命令を取出し，実行する．しかし，プログラム・カウンタはその命令が飛越し命令以外は修正されない．）いかにしてこれらの特権命令が最初に仕立てられるのか疑問が残る．1つの解答は，Multics におけるリンクのための機構を使うことである．特権命令を格納する記憶場所には，その記憶場所を最初に実行するシステムに飛越しが生じるようなコードを入れておく．システムはその試みの正当性を検査することができ，もし許されるならば特権命令を仕立てる．

それゆえ，システムはプロセスが実行できる命令について完全な制御権を持つことになる．プロセスがどのような動作を行う必要があるか，あるいはシステムの呼出し毎に付随する特別の検査を避ける程度に応じて，プロセス毎に異った特権命令を利用させることができる．システムが実行専用である記憶領域の部分に適当な I/O 命令を置くことによって，プロセスは補助記憶の領域を直接アクセスすることさえできる．この方式は SCC-6700 で使われており，ページのレベルで行なわれている．SCC-6700 におけるページは，読み／書き可能，実行専用，特権命令として実行可能，となり得る．

4.1.4 まとめ

4.1.1 節で列挙した保護に関する問題をハードウェアの助けによって解決するという形で，記憶および制御保護の概念を紹介した．記憶保護は物理空間または論理空間に対して適用できる．論理空間の保護は，動的再配置方式の一部分として容易に達成できる．記憶領域は，読み／書き可能，読出し専用，実行専用というような種々のアクセス・モードに割当てることができる．

復習すると，制御保護はプロセスの実行できる命令の種類に関連する．ユーザ・モードとシステム・モードは，多くの計算機によく見られる．ユーザ・モードでは，実行できる命令に制限がある．一方，システム・モードでは，計算

機内のすべての命令が実行できる．また，モード間の切換えの方法を述べた．プロセスよりむしろ記憶領域に制御保護を適用する方が制御保護を割付ける際に融通性を増す利点がある．

言い換えると，リソースの保護を適用できる場所は2つある；すなわち，(1)リソース自身に対して行なうか，または，(2)リソースへのアクセス経路に対して行なうか，である．種々のリソースへのアクセス経路に対してハードウェアによる保護を行う方がより大きな融通性とシステムの効率が得られるように思われる．

4.2　割込みシステム

4.2.1　概　　要

どのようなリソース共用システムにおいても重要な特徴は，システムの種々のプロセッサおよび機器とコミュニケーションする方法である．この要求を扱うために考え出された1つの機構は，**優先順位割込みシステム**である．割込みの基本的な概念は非常に簡単であるが，第3世代までのほとんどの計算機では現実の形で存在しなかった．CPU は種々の I/O プロセッサを動作させたり停止させたりし，ある種の機器に対しては機器に関する細かな制御も行う．割込みシステムがないと，CPU は機器からの明白な形の信号を受けるまで機器を無視する．

ユーザ・プロセスを扱う同一のプロセッサ群がリソース管理を行う限り，優先順位割込みシステムの開発の動機はおそらく当然のことであろう．あるプロセッサが I/O のないユーザのプロセスを扱い，他のプロセッサが種々のリソース管理を行う BCC システム（3.2.2節参照）のようなシステム構成を開発する場合は，ハードウェアの相互コミュニケーション・システムは簡略化できる．まずリソース共用システムにおいて通常必要とされる相互コミュニケーションの信号形式について調べ，それから優先順位割込みシステムに注目した後に，4.2.4節で BCC 方式について述べよう．

機器が割込み線に信号を出した時に起こる最も単純なことは，CPU が決まった記憶場所へ飛越しそこの命令を実行することである．言い換えると，通常それは割込みを扱うルーチンへのサブルーチン飛越しである．割込み処理ルーチンは，使う必要のあるすべての CPU レジスタ類の内容を一時的な記憶場所に退避し，処理が終了した時にこれらのレジスタの内容を復帰してから割込ま

れたプログラムに戻るために飛越しする.

現在の大部分のシステムは，個々の線あるいはいくつかの線のグループに関連した優先レベルを持つ数本の割込み線を備えている．そのようなシステムでは，たとえば第4レベルの割込みが起こりこのレベルに対応したルーチンが終了する前にこれより高い第2レベルの割込みが起った場合，制御はこのより高い第2レベルに対応したルーチンに移される．第2レベルのルーチンが終了したときに制御は第4レベルのルーチンに戻され，このルーチンが終了すると制御は初めに割込まれたプログラムに戻される.

リソース共用システムにおいて，割込みを扱うプロセスは主要な出発点になる．種々の割込みによって，I/O スケジューリングおよびリソース割付けアルゴリズムが開始したり停止したり，あるいは条件付きで動く．すべての割込みは，システムの処理や手助けを求めるためのシステムに対する明白な呼出しである.

ユーザ・プログラムは，次の3つの状態のうちの1つにあると考えられる．すなわち，(1)実行の用意ができた（準備状態)，(2) I/O 待ち，または他のプロセスの終了を待つ間ブロックされている（ブロック状態)，および (3)実行している（実行状態)，である．システムが割込みを受けると，その結果，1つあるいはそれ以上のユーザ・プログラムならびにシステムの状態が変わる．割込みには主要な2種類がある．すなわち，(1) CPU の外部から起こされる割込み，たとえば，I/O の割込みなどや，(2) CPU 内で起こされる割込み，たとえば，演算のあふれや不当命令の実行などである.

4.2.2 信号の形式

4つの信号形式を簡単に調らべ，それらがシステムやプログラムに及ぼす効果について検討しよう.

I/O 信号　この割込みは I/O 動作の正常あるいは異常終了によって起こされる．このような割込みは，チャネルがデータのブロック転送を完了した時，あるいは機器がテープの巻き戻しやディスク・アームの位置付けのような動作を完了した時に起こされる．割込みレベルはほとんどのシステムで機器要求と同一であるが，信号の正確な意味は機器内の付加状態情報を得てはじめて判断できる．その信号を受取りその意味が決定されると，いくつかのことが起こる．典型的な例は次のようなものである．すなわち，

1. 機器はシステム記憶内の I/O バッファとの間で情報転送の用意がで

きている．

2. 機器が終結し，他のユーザが使えるようになる．1人のユーザの状態をブロック状態から準備状態に変える．

3. システムの注意を必要とする誤まり条件が生じた．

タイマ信号 タイムシェアリング・システムには，一定間隔で割込みを起こすハードウェア・クロックがいくつか付加されている．940のクロックは，16ミリ秒毎に割込みを発生する．(時間々隔の細かいクロックが望ましい．なぜならば，適切な制御のために，しばしば細かな時間々隔が必要であるからである．) これらのタイミング信号は，課金の目的に使われたり，システム保護やスケジューリング・ルーチンへの主要な入力として使われる．システム保護としての使われ方には色々ある．通常，I/O 動作が予期した時間に終了したことや，重要な装置の一部を結合するハードウェアあるいはソフトウェアの故障が起こらなかったことを確認するための I/O 動作の定期的な検査なども含まれる．

ユーザ・プログラム信号 2種類の割込みがある．その1つは演算操作からのもので，割算の検査とか演算のあふれ，アンダーフローというような信号に起因する．他方は不当命令の実行，あるいは不当な記憶場所への参照によるものである．

ハードウェア誤まり信号 これらの割込みはデータ転送で検出された誤まり，または電源異常というようなハードウェアの誤まりに起因する．

トラップ 先に述べた CPU の外部で起こされるものと CPU の内部で起こされる主要な2種類の信号について調べてみよう．その違いはしばしば，**割込み** (interrupt) と**トラップ** (trap) であるといわれる．必ずしも明確でないし，厳密でもあるいは形式化されてもいないが，その違いは設計の問題点の相違を見抜く上で有用である．両者ともハードウェアの効果は同じである．すなわち，プロセッサが決まった記憶場所の命令を実行すべくサブルーチンへ飛越す．マルチプロセッサ・システムでは，その違いをより明確に見ることができる．割込みはプロセッサの外部で起こされる信号である．トラップはプロセッサの内部で起こされる信号である．I/O 機器からの信号は割込みであると言える．タイマ，ユーザ・プログラム信号，およびハードウェア誤りのような信号はトラップであると言える．割込みはシステム内のどのプロセッサによっても処理されうるが，トラップは一般にそれが発生したプロセッサによって処理される．割込みはプロセスが待っていたある条件が満足されたことを示すが，ト

ラップはプロセスをプロセス内のある記憶場所へ飛越しを起こす．設計上の問題におけるこれらの2種類の信号の違いは，マルチプロセッサ・システムでは割込みは入ってきた信号より低い優先順位の機能を実行している最初のプロセッサへ導かれるということである．その時，このプロセッサは適当な割込み処理ルーチンに制御を移す．トラップ信号はそれを起こしたプロセッサにのみ影響を与える．図 4.3 はその考え方を示している．

4.2.3 割込み処理中の課金

割込み処理中の課金は，2つの方法で扱われる．XDS-940 システムでは，割込みが他のユーザのためにサービスしているものであっても，その割込み処理時間は実行中のプログラムのユーザに課金される．一定期間では各プログラムに必要な割込み時間は平均化されるので，この考え方は不当ではない．したがって，割込み処理の課金を分類するのに必要なハードウェア-ソフトウェアに金をかけてもむだである．

一方，Multics システムでは割込み処理時間はプログラムがサービスされているユーザに課金される．

4.2.4 優先順位の割当て

大部分のシステムは，種々の割込み信号に対応した優先順位のレベルをあら

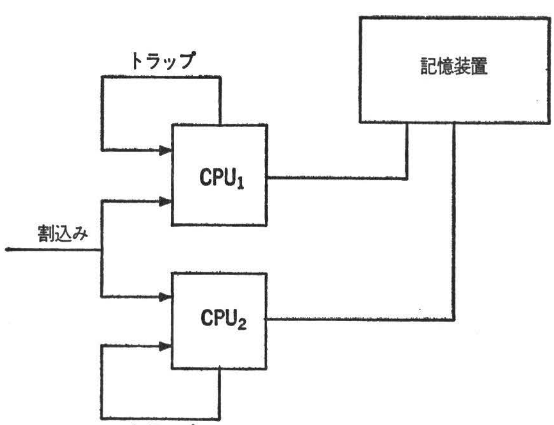


図 4.3 割込みとトラップの違い

かじめ割当てている．これはシステム内の他のプロセスの優先順位と割込みルーチンの優先順位との間に区別を作り出している．ハードウェア割込みルーチンは，自動的にシステム内で実行されている他のプロセスよりも高い優先順位を持つ．このことは，割込みルーチンはシステム内の他のプロセスとは別に扱う必要があるということと，割込みルーチンは他のプロセスのために用意された設備を簡単には利用できないということを意味している．多くのハードウェア割込み信号は，固定された優先順位を必要としない．そのかわりに，実行中の特殊な機能，あるいはシステム状態を基にした優先順位を持つことができる．さらに，ハードウェア信号をシステム内で発生する他の信号とは別に扱うようにすると，システム内に必要な機構の数は増す．理論的には，ハードウェア機器からの信号をソフトウェア・プロセスからの信号と同じように取扱えないという理由はない．

ハードウェアとソフトウェアの信号を統一する1つの方法が，Lampson により提案されている[2]．すなわち，中央プロセッサの使用を待っているプロセスに対する待ち行列の取扱いや，あるプロセスから他のプロセスへの中央プロセッサの切換えができる特殊なマイクロプログラム制御のプロセッサ（4.3 節参照）にスケジューリング機能を持たすことである．この特殊なプロセッサは，存在する割込みシステムのすべての機能を実行する．すなわち，多数の信号を監視し，信号間の優先順位を認識し，そしてプロセッサが信号の示すところの準備状態にある最優先プロセスの命令を実行していることを調べる．最後の機能に関して，プロセッサはあるプロセスから他のプロセスへ高速に切換えることが要求される．

すべてのハードウェア信号は初めはマイクロプログラムされたスケジューリング・プロセッサで扱われる．この特殊なプロセッサは処理中の信号の優先順位を調べる．そしてサービス要求を処理するためにプロセスに対する待ち行列のエントリーを作るか，あるいは優先順位が十分に高いものであれば，その信号を扱うプロセスへ中央プロセッサを切換える．そのような特殊なスケジューリング・プロセッサを使えば，もはや割込みルーチンの概念は必要ではない．そしてすべてのスケジューリングの機能が統一される．

本方式の1つの難点は，アルゴリズムをハードウェアで組むことで設計の変更が難かしくなりそうなことである．容易に書換えできる制御記憶を持つマイクロプログラム可能なプロセッサを使えば融通性が得られる．現在のシステムと同じ程度の融通性はこの方法で得られる．しかしながら，設計やプログラミングの労力という面では修正するための費用はより大きくなる．

もう1つの難点は，統一されたシステムにおいてプロセッサをあるプロセスから他へ切換えることは，現在の割込みシステムで行うよりも多くの状態ベクトルを切換える必要があるということである．第1章で述べたように，状態ベクトルはプロセッサの中核のレジスタ，アドレス・マップ，およびその他の状態情報から成る．種々のレジスタを退避したり再ロードすることは時間の浪費であり，数マイクロ秒以内の応答を必要とする信号を扱う上で困難さをきたす．この問題に対してはいくつかの解決法がある．

1つの方法は，中核のレジスタをごくわずかにし，状態情報の残こされたテーブルへのハードウェアのポインタを保持することである．1つの動作で状態情報を切り換える特殊な命令が可能になる．この種の命令は格納とロードのためにいくつかの記憶サイクルあるいはサブサイクルを必要とするが，命令の取出しのためには1サイクルしか必要としない．他の方法は，XDS Sigma 7システムで行なわれたように中核のレジスタを何組かで持つことである．通常のプロセスは1組のレジスタを共用し，高い優先順位を必要とする少数のプロセスはそれら自身のレジスタ・セットを持つ．残された状態情報テーブルへのハードウェア・ポインタは必要である．

Lampson の提案は少し修正されて BCC システムに実現されているが，そこでは多数の独立したプロセッサが共通の中央記憶装置に接続されており，あるプロセッサがユーザ・プロセスを扱い，他のプロセッサが特殊なリソース管理を行う（3.3.2 節参照）．BCC 方式では，プロセッサへの割込みは存在しない．他のプロセッサが与えられたプロセッサの**アテンション**（attention）を得ることのできる1本のアテンション線だけがある．I/O または外部記憶機器を扱うプロセッサのために，通信制御装置のような接続された機器がそれを制御しているプロセッサのアテンションを得ることのできる別の線がある．これらの線は割込みを起こさないが，プロセッサ内の**アテンション・フラグ**（attention flag）（フリップ・ロップ）をセットする．これらのフラグはプロセッサの都合の良いときに調べられる．

プロセッサはアテンション・フラグがセットされているのを発見すると，共通の主記憶内のテーブルへ行くか，あるいは機器の状態語を入力する．そのテーブルにプロセッサあるいは機器の識別子やコミュニケーションすべきその他の情報を格納する．ここまでくると，BCC システムは初期の計算機に見られるようなコミュニケーション機構の形式を使っているのと同じことになる．BCC の設計者は，このように構成したシステムでは優先順位割込み機構は不必要であると思っている．なぜ優先順位割込み機構が不必要であるかを知るた

めに，BCC システムをさらに詳しく調べてみよう．

まず，ユーザ・プロセスを扱うプロセッサについて考察する．これらのプロセッサは I/O や補助記憶機能制御は行なわず，限られたリソース管理機能を実行する．これらのプロセッサが必要とする主要信号は，プロセスを切換える信号である．しかしながら，これらのプロセッサは他のリソース管理プロセッサとコミュニケーションしなくてはならない．プロセスを切換える信号は，プロセスを切換えるために必要な情報を見つけ出すために特定の記憶場所に飛越しを起こさせる割込みに似ている．

ユーザ・プロセスの1つがリソース管理プロセッサに援助を要求するときはメッセージを主記憶に置き，適切なリソース管理プロセッサへのアテンション線に信号を出す．呼び出しているプロセッサはそのサービス要求の性質によって，要求が完了するのを待つか，あるいはプロセスを切換える．プロセッサは数マイクロ秒以内で完了するような要求に対してのみ待つ．

リソース管理プロセッサは，現行のタスクを完了したとき，さらに行うべきタスクがあるかどうかをみるためにアテンション・フラグを検査する．もしフラグがセットされていれば，記憶装置内にあるプロセッサ間のコミュニケーション・テーブルで示される最優先タスクを取り出す．

同様の方法で，リソース管理プロセッサは接続されている機器に対応するアテンション・フラグを調べることができる．ドラムやディスク等の補助記憶機器以外の機器に対する I/O は直接 I/O 命令によって行う．補助記憶リソース管理ルーチンは，ドラムやディスクの転送のための特殊な**高速直接記憶アクセス・チャネル**(DMA)を制御する．リソース管理プロセッサの各タスクは極めて簡単なものであるから，アテンション・フラグを検査するための時間は正確に決っている．そしてリソース管理プロセッサは高速なので，上で概説した簡単な相互コミュニケーション機構による遅延は重大にはならない．読者は第5章を読んだ後で上の議論を再考してほしい．そこでは，スケジューリングについてより詳細に議論し，ハードウェアおよびソフトウェア信号の処理を統一する問題について詳しく述べている．

現在のシステムは，割込みレベルを選択的に切り換えたり，保存したり，一時的に無視したりできるようにすることで，優先順位の調整を行なう機構を作り出している．あるレベルの優先順位を高めるために，割込みシステムにある種の命令を出すことができる．この命令は割込みシステムがより高い優先順位の割込みが入ってきても一時的に蓄えて，システムに入らないように飛越しを起こさせる．優先順位調整に対するこの方法はむしろ融通性がなく，潜在的に

危険である可能性がある．なぜならば，システムは割込みが無効である間，末処理のサービス要求の存在に気がつかないからである．もし割込みがハードウェアと同様にソフトウェアによって開始できるなら，さらに融通性が得られる．たとえば，XDS-940 では端末から送られる字を失なわないようにするために，通信制御装置の割込みに高い優先順位を与えている．入ってくる字をシステム内部コードに変換したり，それをテレタイプに呼戻したり，それがシステムに対して特に重要な制御用の字であるかを見るための検査や，その字を適当な仕事に対応したバッファに置いたりするために必要なある量の処理がある．もし上で述べた機能のすべてを1つのルーチンで取扱うようにすると，それ以外の同じか低い優先順位の割込み信号は，その全処理時間中無視される．ただ時間的に問題がある動作は記憶装置に字を移すことだけで，他の機能は低い優先順位のものである．この問題をシステムで扱うには，高い優先順位の割込みルーチンの一部として端末の識別子を付けて字を記憶装置に移し，その後ソフトウェアで低い優先順位の割込みを開始し，高い優先順位のルーチンから抜け出す．低い優先順位のルーチンは残りの機能を行うが，高い優先順位の信号によって割込まれることがある．

別の方法が，Digital Corporation の PDP-10 システムでとられている．いくつかの割込み優先順位のレベルがあり，各レベルは各機器を通過している I/O バスの1本の線で表わされる．各機器はソフトウェアによって優先順位が割当てられる．そのソフトウェアはどの線に割込みの信号を送るべきかを示す．この方式は機能とシステムの状況に基づいた優先順位を割当てる上でシステムの設計者に融通性を与える．

4.2.5 まとめ

タイムシェアリング・システムには，融通性のある可変優先順位の割込みシステムか，あるいはプロセッサ相互および機器間でコミュニケーションするための適切な方法が必要である．アテンション信号は，リソース割付け（スケジューリング）・プロセスへの主要な入力となる．スケジューリング・ルーチンの入力はハードウェア信号だけではない．ハードウェア信号をソフトウェア信号と異なった方法で取扱うには，システム設計において余分な機構が必要である．全体のリソース管理機能に注目して，ハードウェア割込みシステムの設計を再検討すべきであると思われる．

4.3　マイクロプログラミングの概念

4.3.1　制御に関する問題の再検討

ハードウェアおよびソフトウェアは不明確で常に変動する領域である．プロセッサ制御装置を設計する際にしばしば用いられ，一般性を増す技術に**マイクロプログラミング**（microprograming）と呼ばれるものがある．これは，ハードウェアとソフトウェアの領域をさらに不明確にする可能性をもっている．マイクロプログラミングという用語には，一般に認められた定義はない．本書の以下の議論でこの用語を使うためにここで1つの定義を与えるが，それは次に示す意味に限定する．この用語は，1951 年に M. V. Wilkes が操作コードをプログラマが定義できる計算機を議論した中で考え出されたように思われる[7]．その時 Wilkes はおそらくそのような計算機の必要はないであろうと予測し，1958年に彼の考えを再確認している[8]．しかしながら，その時以来ハードウェア技術が変化し，オペレーティング・システムやその他のシステム・ソフトウェアが複雑になるに至って，再びこの問題が持ち上がってきた．

本節では，マイクロプログラミングの概念を紹介し，システムの設計者に対するその掛かり合いを簡単に示すことにする．この技法の利点を読者が関係している特定のシステムに照らして評価することができよう．本節の題材は，Flynn と McLaren の有名な論文[11] を参考にしている．

初めに制御に関する問題の性質を再検討する必要がある．図 4.4 は制御の観点から見たプロセッサを示している．プロセッサの基本的なリソースは，レジスタ群とその内容，実行論理回路，および主記憶へのアクセスである．基本的な制御の問題は，これらのリソースを各命令の操作コードで指定された機能を実行するように割付けることである．実行制御装置はレジスタの内容に対して通常の組み合わせ論理（時間に関係なくそれ自身の記憶をもたない論理）を行うものである．命令の実行を行うために次の機能が必要である．すなわち，

1. プロセッサ・レジスタと実行制御装置間のデータの流れを制御する機能
2. 実行制御装置が渡された演算数に対してどの論理機能を実行するのかを制御する機能
3. 機能1および2を何回か繰り返してシークエンスを作る機能

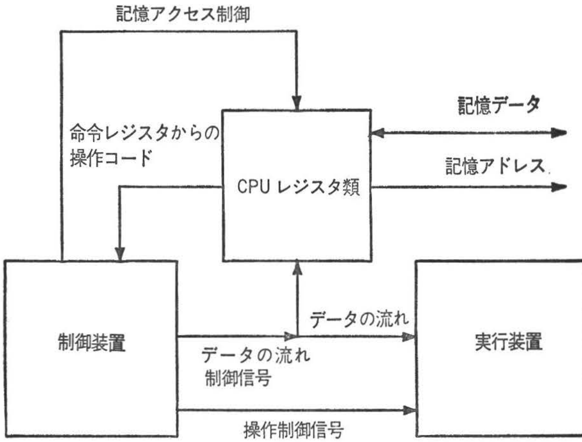


図 4.4 制御の観点からみた CPU の構成

言い換えると，1命令で指定される操作は，プロセッサの基本的なハードウェア論理によって定義される1つかそれ以上の基本操作から成っている．これらの基本的な操作は，**マイクロステップ**（microstep），あるいは**マイクロ操作**（microoperation）と呼ばれる．1命令の実行を指定するのに必要なマイクロステップの集合は**マイクロプログラム**（microprogram）と呼ばれる．1つのマイクロステップは，ある数Nのゲートの状態によって完全に定義される．この数はプロセッサによって，数10から数1000まで変化する．しかしながら，多くの操作はレジスタの内容全体に対して働くので，実際には各マイクロステップを指定するのにより少ない実際のゲート操作の記述で良い．

1命令の操作コードは，特定の操作を行うために必要な組合せ論理と順序論理をコード化したものである．各マイクロステップの実行時に，制御装置はそのマイクロステップが指定するゲートに信号を送らなくてはならない．ある種の機構が要求されたステップを通るように制御装置を順序付けるために必要である．標準の制御装置を実現するためにカウンタが使われる．カウンタの状態と命令の操作コードがマイクロステップ指定する．

復習すると，プロセッサの基本的な論理レベルの操作は，レジスタから実行制御装置に情報を入れること，実行制御装置に渡された演算数の単純な組合せ論理的な変更を行うこと，および結果をレジスタに戻すことである．同じような段階がアドレス計算，および記憶装置をアクセスするために必要である．マ

イクロプログラムの概念を説明する目的のために，これらの段階を無視し，命令実行中の実行制御装置の動作だけを考えることにする．

4.3.2　制御装置の設計におけるランダム・アクセス記憶の利用

これらの制御機能は制御装置の一部に高速のランダム・アクセス記憶機器を使って楽に取扱うことができる．価格および性能に関する問題は後で検討する．各マイクロステップを指定するのに必要な組合せ情報は，記憶装置の1語で与えられる．順序情報は取出す語の順序で与えられる．各語は**マイクロ命令**と呼ばれる．操作コードで指定された通常の命令の操作を行うマイクロ命令の順序を**マイクロプログラム**と呼ぶ．このランダム・アクセス記憶を使うこととマイクロプログラミングの概念を関連付けたい．標準的なマイクロ命令は，図4.5に示すように構成される．**ゲート操作指定部**は実行すべき現在のマイクロステップを指示する．**テスト・コード部**はある条件を示し，その条件が“1”ならば，次のマイクロ命令は次の**命令語アドレス部**で与えられる制御記憶の場所から取出される．条件が“0”であれば，次のマイクロ命令はその命令語の次のアドレスから取出される．テストされる状態とは，たとえば，単純な加算のあふれ，結果の符号，あるいは種々の信号の組合せなどである．

1つの命令のための必要なシークエンスにおける最初のマイクロ命令のアドレスは操作コードあるいは操作コードの関数によって与えられる．この説明は非常に簡単であるが，その基本的な概念を示している．もっと凝った方法も考えられているが，それは一時記憶のアドレス可能なビットや語を使ったり，制御装置内にあるマイクロインデックス・レジスタを使ったりしている．またマイクロ命令内の別のアドレスをいくつか使ったりしている．さらに実行装置の現在または過去の状態に基づいて次の命令語アドレスを作り出す方法もある．経験によると，特に有益な性質はマイクロプログラム・サブルーチンの機能である．選択可能な種々の設計パラメータは，一般に制御記憶の価格，要求される性能，実現される命令の形式，およびプロセッサのデータ経路の配置などに基づく[9]．マイクロプロセッサによって行なわれる操作は，マイクロプログラムよりも高いレベルのプロセッサが行うものと概念的に同じである．主な違い

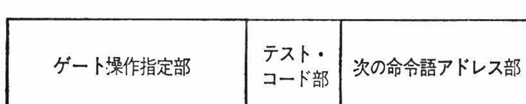


図 4.5　典型的なマイクロ命令語

は，マイクロプロセッサはデータとして内部の論理条件を使い，アドレス論理信号に影響を与え，プロセッサのデータ経路やレジスタに対して細かい制御を行うことができることである．

適当な性能を得るためには，制御記憶の記憶アクセス時間は，レジスタ内容を実行装置に入れ，組み合わせ機能を実行し，再びレジスタに結果を戻すために必要な時間の約半分でなくてはならない．この性能要求を経済的に満足するために，制御記憶を実現する記憶技術は，過去，読出し専用記憶を選んできた．比較的安価で高性能の集積回路記憶が出現したので，制御機能のために読み書きできる記憶装置を経済的に利用できるようになった．必要な記憶装置の大きさはプロセッサの設計に関係するが，おそらく中型から大型の計算機で，1語約100ビットで約1000から2000語であろう．マイクロプログラムの有効な利用のためには，主記憶と制御記憶のアクセス時間に4倍から10倍の速度の差が要求される．この速度差で，主記憶の1サイクルの間に数個のマイクロ命令が実行できる．

4.3.3 マイクロプログラミングに含まれた意味

上述の議論の意味することは明らかである．マイクロプログラミングに基づいた計算機を使って仕事をするソフトウェア設計者には，その計算機の構造を指定する際に大きな自由度が与えられる[11]．仮にアセンブリ言語プログラマが使える命令セットを**マクロ命令**（macroinstruction）と呼ぶならば，ソフトウェア設計者は自分のマクロ命令セットをもっと自由に選ぶことができる．なぜならば，基本的なマイクロプログラミング論理およびその装置が一旦実現されると，新しいマイクロプログラムを書くことによって新しい命令を作ったり，命令を修正することが容易にできるからである．リソースおよび割付け待ち行列，字の操作を取扱う基本的なリスト処理やテーブル操作などのオペレーティング・システム内で必要な多くの頻繁に使われる操作は，マイクロプログラムによってうまく設計されたマクロ命令に仕立てることができる．言い換えると，マクロ命令は設計者が希望するデータ構造に向くように実現できる．普通のサブルーチンのかわりにマイクロプログラムのシークエンスを使えば，記憶サイクルを節約することができる．

頻繁に使われる重要な機能をマイクロプログラムのシークエンスで実現した良い例は，IBM 360/50 に付け加えられた評価（EVAL）命令で Rush Time-sharing System で使われている[10]．この命令は，標準的な形式の PL/1 表現をハードウェアで直接的に評価するものである．オペレーティング・システム

の中にもマイクロ命令のシークエンスに完全に移せるような頻繁に使われる部分がある．たとえば，2.5.8節で述べたようなSCC-6700連想マップを取扱う操作である．そのほかにも，割込みを処理するルーチン，プロセッサの一部，および主記憶や補助記憶の管理機能を行う部分などがその例である．

システム設計におけるマイクロプログラミング技術の他の領域は，補助記憶の割付けとアクセスといった機能に向いた特殊プロセッサの領域である．特に主記憶を拡張するために必要な機能である．3.3.2 節で述べたこともその例である．また，6.1.3 節で概説するのもその例である．4.2.4 節と 5.2.8 節で述べたように，Lampson はプロセッサの割込み処理やスケジューリング機能の全部をマイクロプログラム化したプロセッサに集積することを提案した[2]．もしマイクロプログラムが容易に変えられるようになっていれば，違った割付けアルゴリズムやパラメータを実験するための融通性がさらに得られる．

マイクロプログラムを書くことは，普通のプログラムを書くことより難しい．なぜならば，非常に詳細なハードウェアの知識が必要であり，また多数の状態が1つのマイクロ命令によって制御され，テストされるからである．マイクロプログラムを使用する動機の1つは効率にあるので，ALGOL あるいは FORTRAN などの高いレベルのマイクロプログラミング言語の利用は無理であろう．しかしながら，プログラミングの基本的な概念をマイクロプログラミングでも使っている．

参　考　文　献

1 **Graham, R. M.:** Protection in an Information Processing Utility, *Commun. ACM*, vol. 11, no. 5, p. 365, May, 1968.

2 **Lampson, B. W.:** A Scheduling Philosophy for Multi-processing Systems, *Commun. ACM*, vol. 11, no. 5, May, 1968.

3 **Lichtenberger, W. W., and M. W. Pirtle:** A Facility for Experimentation in Man-Machine Interaction, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, 1965.

4 **Gibson, C. T.:** Time-sharing in the IBM System/360: Model 67, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 28, p. 61, 1966.

5 **Glaser, E. L., et al.:** System Design of a Computer for Time-sharing Applications, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 197–202, 1965.

6 **Evans, D. C., and J. Y. LeClerc:** Address Mapping and the Control of Access in an Interactive Computer, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 23–30, 1967.

7 **Wilkes, M. W.:** The Best Way to Design an Automatic Calculating Machine, *Manchester Univ. Computer Inaugural Conf., Manchester*, July, 1951, pp. 16–18.

8 Wilkes, M. V.: Microprogramming, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 12, pp. 18–19, 1958.

9 Tucker, S. G.: Microprogram Control for System/360, *IBM Systems J.*, vol. 6, no. 4, pp. 222–241, 1967.

10 Babcock, J. D.: A Brief Description of Privacy Measures in the RUSH Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 301–302, 1967.

11 Flynn, Michael J., and Donald M. MacLaren: Microprogramming Revisited, *Proc. ACM Natl. Conf. 1967*, pp. 457–463.

12 Lampson, B. W.: Scheduling and Protection on Interactive Multi-processor Systems. *Univ. Calif. Berkeley Project Genie Doc.*, 40.10.150, Jan. 20, 1967.

第 III 部

ソフトウェアの概念

第 5 章

プロセッサおよび記憶の割付け

5.1 ソフトウェア概念の概要

ソフトウェア・システムはユーザの要求を満足させるようにシステムのハードウェア・リソースの割付けの仕事を行うのみならず，それ自体が重要なリソースの集合を含んでいる．ユーザはソフトウェアの窓を通してシステムを眺め，直接ハードウェアにアクセスするよりはるかに容易に多くのシステム機能を利用することができる．

ユーザはファイル・システムをアクセスするが，ファイル・システムはユーザのコンソール使用中の仕事を次にコンソールを使用する時まで貯え，ユーザが開発中あるいは使用中のプログラム間のコミュニケーション・リンクとして働く．そして，ユーザは他人との共同作業を許されることから，その人達の労力に頼ることができる．おそらくファイル・システムはユーザの観点から見て最も重要なシステム要素であろう．

ユーザは言語翻訳ルーチン，テキスト・エディタ，デバッグ・ルーチン，ライブラリ・ルーチンおよびシステム機能にアクセスすることができる．このようなソフトウェア・サブシステムや補助ルーチンはユーザが興味を持っている情報処理の仕事を遂行するためにハードウェアにコマンドを出すユーザの作業を簡単にする．

ユーザはコマンド・システムを使うことができ，それはユーザの要求を満足させる過程においてシステムに指令を出すのを容易にしている．このソフトウェア・システムはユーザとシステムとの主要な接触点の1つとなっている．

経済性のために多くのユーザ間で全体のシステムを共用させる技術的な問題を解決する際，設計者がかかわりあうソフトウェア・システムの部分は，一般にそれが働き，ユーザの情報処理の要求を満足している限りでは，ユーザにはほとんど直接的に興味はない．ユーザにとってシステムはブラック・ボックス

である．システムはサブシステム，コマンド言語，ファイル・システムやその他のソフトウェア・ルーチンを通して，ユーザがハードウェアを直接アクセスすることにより簡単には得られないような機能をユーザに提供する．ユーザはただ単にその他の多くの人達とリソースを共用することによりそうでない時には得られなかった能力を利用できると知っただけでも，そのブラック・ボックスのリソース共用に関する内部的働きにありがたみを感じる．

システム設計者がこれらの点を心に留めておくことは有益である．設計者の役割はサービスを行うことである．設計者の技術および知識はその目的に対してのみ有用なのであって，技術や知識自体のためではない．おそらくユーザは設計者が誇りに思い，発表しようと計画している巧妙なN次元の web-threaded スケジューリング・アルゴリズムに対しては少しも興味を示さないだろう．特にユーザにとって1次元のアルゴリズムが働き，同じ機能を提供し，同じ価格ならばなおさらそうである．設計者にとって各設計段階において与えられた技術水準に可能な設計上のつり合い問題に気を付けながら，目標に至るまで概念的に簡潔性や一貫性，均一性を維持することは良い鍛練になると思われる．ユーザは自分の情報処理に対する要求，秘密や言頼性に対する要求およびシステムとの容易なインタフェースなどについては常に心に留めておかなければならない。これらの点はここで強調しておきたい．なぜならば，ソフトウェア・システム設計者はハードウェア設計者よりもずっと少ない制約の中で働いているからである．しかしながら，その結果はしばしば貧弱なソフトウェアや人間がいづらいものが作り出されている．これらの言葉に関連して，ここでソフトウェア・システムについて論じよう．

ソフトウェア・システムはプロセッサおよび記憶割付け，ファイル・システム，I/O の取扱い，コマンド・システムやコンパイラ，テキスト・エディタ，デバッグ手段などのサブシステムといった見出しで，自然に論じることができる．これは本書のこの部で用いる方法である．サブシステムについては，どのようにそれらのシステムの残りの部分と接続するかについて簡単に示す以外は論じない．上述の機能的な領域は次の2つのサブクラスの機能に認識できる．(1) 頻繁に使われるか，あるいはシステム・リソースに対する要求の変化に敏速に応答しなければならないので，固定的に主記憶内に駐在していなければならない機能と，(2) システムの一部としてあるいはユーザ・プロセスの一部として主記憶からスワップ・イン，スワップ・アウトされる機能である．

常駐機能はI/O機器のサービス，プロセスのスケジュール，記憶およびI/O機器の割付け，端末とのコミュニケーションの取扱い，非常に高い頻度で起こ

るタスク，あるいは十分速い応答を要求するタスクを取扱う．これらの常駐ルーチンに関連するものは，ユーザ・プロセスおよびシステム・リソースについての状態情報を含む多くのテーブルである．

スワップ可能な機能はあまり頻繁に必要としないサービスに関連し，サブシステムの割付け，システムに対するユーザのログ・イン，ログ・アウト，課金，ファイル・システム操作，およびシステムの敏速な注意を必要としないその他のリソース割付け機能等がある．技術的な設計上の問題の1つは，これら2種の機能の境界を決定することと，ユーザの本質的要求を満たすと同時に常駐機能のために必要な主記憶領域の大きさを最小にすることである．

オペレーティング・システムの実現化の詳細を見ると，システム・データ・ベースを取扱う一般にリエントラントなルーチンの集合から成っていることがわかる．システム・データ・ベースは種々のテーブル，待ち行列，バッファ，およびリスト構造から成っており，それらのいくつかは主記憶内に永久的に駐在し，他は主記憶と補助記憶間を移動できる．ハードウェア設計者がシステム設計を機能ハードウェア・ユニットに変えるために，論理モジュール，フリップ・フロップ類，レジスタ類および制御信号の相互接続の方法を理解する必要があるのと同様に，いかにして上述したデータ構造の種々の基本的形式を生成し，取扱うかを理解するのはシステム・プログラマの仕事である．いくつかのデータ構造の基本的形式については論議の適当なところで招介する．

第3部は，信頼性，保守性，回復性，測定および評価について，主要な機能であるプロセッサおよび記憶の割付け，保護，ファイル・システム，I/O，コマンド・システムと同様に取り扱う．前者の話題は後者と同様に設計を成功させるために極めて重要であり，設計の初期段階から考慮しなければならない．これらについては，本章および後に続く章で若干詳しく考察する．

5.2 プロセッサの割付け

5.2.1 概　　要

中心となる問題はプロセッサの割付けである．システム中に存在するハードウェア・プロセッサの数だけのプロセスしか同時には実行できない．ハードウェア・プロセッサはユーザの望む応答性を満足し，他のリソースの均衡のとれた利用率を達成するように，プロセスに割当てられなければならない．新しいプロセスが実行されようとする時には，主記憶の割付けが行なわれる．マルチ

プログラミング・システムでは，いくつかのプロセスは同時に主記憶内に駐在し，プロセッサと記憶の割付けは相互関係があり，設計上独立に扱うことはできない．上述の問題を解決するための骨組は，第1章で招介したプロセスと仮想プロセッサの概念のまわりに集中している．この時点で1.5節を復習することは有益である．

おおざっぱに述べると，基本的な考えはソフトウェアによって与えられたハードウェア構成を多くの仮想プロセッサに変換することである．各々の仮想プロセッサは自分自身の記憶空間，命令セットを持ち，I/Oやファイル・システムをアクセスし，他の仮想プロセッサと交信したり，記憶装置を共用できる．プロセスはシステム，あるいは他のプロセスがシステムを呼出すことによって，生成されたり消滅させられる．プロセスと仮想プロセッサは，以下に詳しく述べる**文脈ブロック**(context block)，あるいは**状態ベクトル**(state vector) と呼ばれる情報の集合により定義される．

まず最初にある基本概念を明らかにするために，物理記憶空間が無制限であるという仮定の下で，プロセスへの物理プロセッサの割付けを議論しよう．その後，物理記憶空間に制限がある時の記憶とプロセッサの割付けに含まれる問題を議論する．物理的リソースの割付けに深く関係して，相互コミュニケーションの本質的問題がある．以下に述べるプロセッサの割付けについての議論は，Saltzer[1] や Lampson[2] のすばらしい仕事に強く影響を受けている．

5.2.2　基本的なプロセス相互コミュニケーション機能

本質的な相互コミュニケーションの問題を理解するために，まず最初に記憶装置を共用しながら2つのプロセスが2つの別々なプロセッサで実行されているという簡単化した場合を考えてみよう。以後，プロセスAはプロセスBがすべき仕事を**仕事待ち行列** (work queue) に入れると仮定する．その一例は，刻々出力すべき字をバッファに入れる計算プロセスがあり，それらの字を別のプロセスが出力する場合である．プロセスBが待ち行列中にあるすべての仕事を完了したとき，そのプロセスは何をすれば良いのであろうか．待ち行列を見ながらループし，待ち行列中に仕事がなければ再びループする．プロセスAが仕事を待ち行列中に入れた時は，プロセスBは直ちにそれを発見し，適当な機能を実行する．待ち行列を調べるためには少なくとも数記憶サイクルを使うことが必要である．記憶サイクルを使うことは潜在的なシステムの浪費を引き起こし，後に出てくる場合にはさらに深刻になり，プロセッサを拘束してしまう．したがって，その問題に対する良い解答はプロセス自身を新しい仕事が到

着するまで**ブロック**（block）させておくことである.

ブロックするということは次の仕事が到着するまで実行をやめる**機能**である. 2個のプロセスと2個のプロセッサの場合では，プロセスBは**停止命令**を実行するか，あるいはこの機能を実行するためにシステムを明白に**呼出すこと**によって自分自身をブロックできる. 後でわかるように，プロセスが**ブロック**する時はそのプロセスから他のプロセスへ切り換える良い機会である.

新しいタスクが待ち行列に加えられる時に，プロセスBはある機構に**よって**ブロックを解かれるか，あるいは**呼び起こさ**れなければならない. プロセスAは，プロセスBが呼び起こされるべきであることを示すために，システムに対して信号を送る. この呼び起こし機能は，システムが用意すべきプロセス間コミュニケーション機能である. 2つのプロセスが2台のプロセッサで実行しているという単純な場合では，この信号は停止状態への単純な割込みになる. そのように，プロセスは**実行状態**（running）あるいは**ブロック状態**（blocked）の2つの状態の内の1つにあることがわかる. 命令を実行している時は，プロセスは実行状態にあり，実行を再び行うように**呼び起こし**（wake-up）信号を待っている時は，ブロック状態にある。さらに2つのプロセスが互いに仕事を行うためには，データや制御情報を相互にコミュニケーションできなければならない.

情報を失なわないために，プロセスの実行中に生じる呼び起こし信号は保存されなければならず，プロセスがすべての呼び起こし信号が処理され終るまでブロックされないようにしなければならない. このような機能はハードウェア割込みシステム，あるいは相互コミュニケーション・システム中に設けられ，各信号は到着するとそれが認識されるまで保存される. 同じような機能がソフトウェアの相互コミュニケーション信号を取扱うためにも用意される. これらの信号が保持されないと**競走**（race）状態が生じ，その場合には最初の信号は処理されてもその処理中に到着した信号は失なわれてしまう. Saltzer はこのような呼び起こし信号を保存する機構を**呼び起こし待ちスイッチ**（wake-up waiting switch）と呼んでいる[1].

2プロセス・システムでは，1つのプロセスから出された呼び起こし信号は常に他方へ送られ，その信号の行先を示す明白な形のアドレスは必要がない. この概念を多数のプロセスを含む状況に一般化するためには，呼び起こし信号を受取るプロセスを識別する**プロセス識別タグ**（process identification tag）が必要である.

ここでまとめると，プロセスは**実行状態**（running）あるいは**ブロック状態**

(blocked) の2状態のうちの一方にあることになる．ブロックされたプロセスはそのブロック状態を解くためのある種の信号が到着するまで実行することはできない。これらのブロック状態を解く信号を**呼び起こし**信号と呼び，プロセスの状態をブロック状態から**実行できる状態**に変える．あるプロセスのブロック状態を解く信号は他のプロセスをブロック状態にする可能性もある．これらの信号の出所はさまざまである．システム・プロセス，ユーザ・プロセス，ハードウェア割込み，端末コミュニケーション機器，タイマ，ディスクのシーク完了，不当記憶アドレスで命令を実行した場合などである．

呼び起こし信号は関連した状態情報と一緒に，しばしば**スケジューラ**(schedular)[2] と呼ばれるシステムの一部に送られる．Multics システムでは**トラヒック・コントローラ** (traffic controller) という用語を使っている[1]．利用可能な機能の範囲の他に，マルチプログラミング・システムのクラスの差異を定めるものの1つに，スケジューラの設計違いがある．オンライン・ファイル管理システムや他の I/O 専用システムにおいては，主要な呼び起こしは I/O 機器に関連しているのに対し，あとでわかるように汎用タイムシェアリング・システムでは，クロックから出る信号が非常に重要になる．システムという観点からすれば，プロセスの歴史は実行とブロック期間の交番である．各実行期間はこのプロセスをブロックする信号により終結される．このブロック信号をプロセス自身が出すかも知れないし，システム・プロセス，あるいはその他のユーザ・プロセスが出すものかも知れない．

簡単な例は，プロセスが I/O 機器から字の入出力を要求する場合である．もし字が有効でなければ，そのプロセスは字が到着するまでブロックされる．字に対する要求は要求された I/O 機器からバッファを補充するためにシステム・ルーチンを呼び起こす．

種々の特定のリソースの取扱いについて論議する中で，ブロック信号や呼び起こし信号の例を補足した．1つの基本的な設計上の問題は，ブロック信号や呼び起こし信号のコミュニケーションを適切に取扱うことにある．

5.2.3　スケジューリング

概　要　実行できるプロセッサの数よりもプロセスの数の方が多い場合は，あるプロセスはブロックもされていないし，実行されてもいない．この状態を**実行準備** (ready-to-run) あるいは**準備状態** (ready state) という．それゆえに，3つの状態，すなわち，実行状態，準備状態，ブロック状態，があることになる．いくつかの実行準備状態にあるプロセスの内，どのプロセスを次

に実行するかを決める何らかのアルゴリズムが必要である．このアルゴリズムは，しばしば**割振り**（dispatching）あるいは**スケジューリング**（scheduling）**アルゴリズム**と呼ばれる．割振りアルゴリズムはスケジューラの中でも重要な部分であるが，先にみたように，決してスケジューラ全体にはなっていない．

割振りアルゴリズムの設計に対する有用な基準は[2]，(1) プロセスを切換える際にシステムが費やす労力を最小にすること，および，(2) 最も重要なプロセスが即座に実行できるようにするため実行準状備態のプロセスの集合の変更をす早くできることである．

割振りアルゴリズムが上述の基準を満たすためにどのような特徴があるかについて論議した文献は数多く出ている[3]．システム・リソースに関して責任を負いかつ高価でない割振りアルゴリズムの設計目標は，しばしば矛盾をもつ，それゆえに，両方の領域のつり合いを保つことが実際のアルゴリズムに要求される．割振りアルゴリズムについてはさらに5.2.6節で述べる．本章においてはある概念をより大きなレベルの詳細さでくり返して論じる．

準備リストおよびプロセスの優先順位　いかなる割振りアルゴリズムにも基本となるものは，実行準備状態にあるプロセスを記録する方式である．そのようなプロセスのリストを**準備リスト**（ready list）と呼ぶ．ここでプロセッサ割付けの一般的な骨組を調べてみよう．プロセスに送られる呼び起し信号は，そのプロセスを準備リストに入れる．プロセスがブロックされると，ハードウェア・プロセッサを準備リストにあるプロセスに一時的に切換えることができるようになる．割振りアルゴリズムは準備リストにあるプロセスのどれを実行させるかを選択しなければならない．実行するプロセスの順位を実際に決める機会は2度ある．すなわち，(1) プロセスが準備リストに置かれるとき，と (2) プロセスが準備リストから取り除かれるときである．

選ばれた機構は別の割振り方法を探究できる融通性を持ち，そして同時に次に実行するプロセスの決定を行なう際にほとんどシステムの努力を必要としないようなものでなくてはならない．次に何をするかを決めるのに計算時間のかなりの部分，たとえば20%，も必要とするスケジューリング機構は，どのようなものであっても，おそらく受け入れられないだろう．

割振るためによく使われる一般的な方法は，**優先番号**（priority number）にしたがってプロセスの準備リストでの順位を決めるものである．したがって割振りルーチンは単純であり，次に実行するプロセスは最優先のものから選

ぶ．ここで，プロセスの優先順位を取り扱うのに有効な3つの一般的な方法についていて述べる．

最も単純な機構は，**先着順サービス**(frst in, first serve)による優先順位付けである．プロセスが呼び起こされると，準備リストの最後に置かれる．割振りルーチンは常に次に実行するプロセスをリストの先頭から取り出す．このようにして，優先順位は準備リスト中に入れられていた時間にともなって増加する．

2番目の技法は，**固定優先順位技術**（fixed-priority technique）と呼ばれるものである．この方式では，プロセスは生成された時に固定した優先番号が割当てられる．プロセスが準備リストに置かれる場合に，それより低い優先順位のものより前で，それと等しいかあるいは高い優先順位のものより後に置かれる．この方式でも割振りルチーンは次に実行するプロセスをリストの先頭から取り出す．これら両方の方法とも単純であり，システムはスケジュールする際にはほとんど手はかからないが，ユーザに対するより良い応答性とシステム・リソースの均衡のとれた利用率をもたらすようなシステム情報に対しては融通性がなく感度が悪い．

3番目の方式は**可変優先順位技術**(variable-priority technique)と呼ばれ，プロセスを準備リストに入れる時に計算した優先順位を割当てる．優先順位の計算には，設計者が判断できるような種々の情報，たとえば，過去に使ったプロセッサ・リソースの量，必要な記憶装置の大きさ，ブロックされた理由，その他のあらゆる情報を考慮に入れることができる．その概念は各プロセスが別々の優先順位割当て手順，すなわち個別スケジューラを持つというように一般化できる．もし各ユーザが自分に最も都合の良いようにスケジューラを書こうとすれば混乱が起こるので，個別スケジューラはシステムが用意する．考え方は実時間操作を制御しているプロセスに対しては常に準備リストの先頭に置くようにその優先順位を計算し，一方その他のクラスのサービスに対しては別の方法で計算した優先順位を持つようにすることである．たとえ計算による優先順位方式でも，もしプロセスが優先順位にしたがって準備リストに順序付けられるならば，割振りルーチンは新しいプロセスを実行させる時にはリストの先頭にあるプロセスを取出すような単純な手順を行なえば良い．

タイマおよび先取権　　先に考えたことから，2つの複雑な問題が付随して出てくる．すなわち，(1) 実行状態に置かれたプロセスが自分自身でブロックしなければどうなるか，そして，(2) 準備リストに加えられた新しいプロセスがスケジューラによって実行中のプロセスよりも高い優先順位を与えられた場

合はどうなるか，ということである．

最初の問題は一旦プロセスが実行を開始されると，すべてのユーザに対して応答性を保とうとするならば，必要以上に長く実行させるわけにはいかないという事に注意すれば解決できる．それゆえ，スケジューリング信号の重要な発生源の1つは**クロック**源(clock source)，あるいは**インタバル・タイマ**（interval timer）である．クロック源はある決った周期で割込みを発生する機器で，経過時間を記録するレジスタを1加えるのに使われる．同様にインタバル・タイマはセットされると，ある与えられた経過時間後に信号を送る機器で，その時間はプログラム制御の下で可変である．

プロセスがブロックされるまで実行できる時間の長さは**クオンタム**（quantum）あるいは**タイム・スライス**（time slice）と呼ばれる．クオンタムの量はスケジューリング・アルゴリズムの重要なパラメータの1つである[4,21]．それは固定にすることもプロセスの大きさ，プロセスの優先順位，あるいは前回のプロセス実行時間などのパラメータに依存して可変にすることもできる．一般に大部分のプロセスが1クオンタム期間内に処理要求を満足できるようにクオンタム量を調節できれば平均応答時間を減らすことができる．Saltzer が提唱した簡単な例を考えれば，なぜこのことが正しいかを直観的に理解できる．実行すべき5つのプロセスがあり，各プロセスは10秒の計算時間が必要であると仮定する．さらに1つのプロセスから他のプロセスへプロセッサを切り換えるためのオーバーヘッドあるいは損失時間はなく，スケジューリング・アルゴリズムは1つのプロセスを固定のクオンタムで実行させ，次のプロセスに同じクオンタム期間プロセッサを切換え，すべてのプロセスが終結するまでそのようにすると仮定する．

1クオンタムを1秒とした場合にどうなるかを考えてみよう．各プロセスは別のプロセスへプロセッサが切り換えられるまで1秒間計算する．10秒後には各プロセスは2秒間の計算を行ない，45秒後には各プロセスは9秒間の計算を行なったことになる．そして46秒後には最初のプロセスは終結し1秒毎に残りのプロセスも次々に終結する．この場合平均応答時間は48秒である．

ここで1クオンタムを10秒とした場合を考えてみる．10秒後には最初のプロセスは終結し，20秒後に2番目のプロセスが終結し，そのようにしてすべてのプロセスが終結するまで50秒かかる．この場合の平均応答時間は30秒である．クオンタム量をプロセスが終結できる大きさにまで増加すれば，平均応答時間は減少する．実際の状況はプロセッサを切換える際にオーバーヘッドや休止時間があるのでさらに大きく改善されるであろう．

実際には全システム・リソースの均衡をとり，計算向きプロセスの応答時間を極度に増加させず，高度な相互作用を持つプロセスに対して良い応当性を与えたいので，クオンタム量やスケジューリング方法を選択する問題は複雑になる．平均応答時間を最小にすることはそれ自体では十分な基準にはならない．それゆえ，さらに複雑なスケジューリング方法および可変クオンタム量がしばしば使われ，たとえばこれらに関しては5.2.6節で論じる．

インタバル・タイマからの信号で実行中のプロセスがブロックされたとき，それは直ちに実行できる状態にあり，ある優先順位を割当てられて準備リストに戻されなければならない．それからプロセッサは準備リストの最優先プロセスに対して割当てられる．

ここでプロセス識別タグおよび優先番号の他にもう1つの付加的な情報が準備リスト上で必要となる．すなわち，このプロセスがプロセッサの制御を得た時に実行を許されるクオンタム間隔である．各プロセスに対してこのクオンタム間隔を決める設計上の問題が付随的に出てくる．

スケジューラが現在実行中のプロセスよりもさらに重要であると決めたプロセスを準備リストに入れると，実行中の低い優先順位プロセスはブロックされ，上で述べたように準備リストに戻されてプロセッサはもっと重要なプロセスに切り換えられる[1,2]．この**先取権**（preemption）の概念はシステム・リソースの利用率が不均衡にならないように注意深く取扱う必要がある．たとえば，もしスワッピングがあり，重要なシステム・リソースが主記憶にプロセスを持ち込むために使われる場合，スケジューラは持ち込まれたプロセスが直ちにブロックされてしまわないようにブロック信号を遅らせる機能を持っていなければならない．

ま　と　め　　ここでプロセッサのスケジューリングに伴なう主要な概念を紹介した．すなわち，

1. プロセス状態の概念，すなわち実行状態，準備状態，ブロック状態
2. 呼び起こし信号の概念
3. 準備リスト，優先番号，および割振りルーチンの概念
4. インタバル・タイマおよびクオンタム量の概念
5. 先取スケジューリングの概念

である．次にプロセス間コミュニケーション，優先番号，およびクオンタム量についてさらに詳しく述べることにする．

5.2.4 プロセス間コミュニケーション

呼び起こし信号およびブロック信号には2つの基本的な発生源がある．第1の発生源は外部ハードウェア割込み信号で，それらはまず**割込み処理ルーチン**と呼ばれるシステム・ルーチンで取扱われる．割込み処理ルーチンはどのような行動をとったら良いかを決定し，該当するプロセスに適切な呼び起こしおよびブロック信号を送らなければならない．

ブロックおよび呼び起こし信号の第2の発生源はソフトウェアによって内部的に作り出される．たとえば，XDS-940 および Multics システムではプロセスがそれと一緒に仕事をする並列あるいは子プロセスを生成でき，それらは互いにブロックおよび呼び起こし信号を送って**インタロック**できる．XDS-940 では後者の信号は**ソフトウェア割込み**システムと考えられる機構によって実現されている[22]．プロセスは自分自身の割込み処理ルーチンを持ってなければならないが，システムはその信号を伝送する機構を用意しなければならない．ソフトウェア割込みが送られるプロセスに対して状態情報を有効にするために，適当なプロセス間コミュニケーションも必要となる．

現存のシステムではブロックおよび呼び起こし信号のコミュニケーションに対して，2つの主要な方法があるように思える．1つの方法は信号を出すプロセスは影響を与えるすべてのプロセスを明白に知っていなければならない．もう1つの方法は信号を出すプロセスはどのプロセスが影響を受けるかを明白に知らなくても良いが，ただある記憶セルに1を加えるといった単純な動作を実行すれば良い．いずれの方法でもソフトウェアおよび外部ハードウェアで発生する信号を伝達するのに利用できる．

XDS-940 のソフトウェア割込みシステムの種々のものについて考えてみよう[22]．数ビットの情報である**割込みフィールド** (interrupt field) をスケジューラが既に保持しているプロセスに関する情報と関係付けることができる．これらのビットはソフトウェア割込みに相当する．プロセスはスケジューラにプロセス識別タグと割込みをかけるプロセスの割込み番号を送ることによって割込みを伝える．スケジューラは指定されたプロセスの割込みフィールドに適当なビットをセットとして戻る．信号を送っているプロセスはその後実行を続けるかあるいはそれ自身が割込みを受けるまでブロックする．割込みを受けたプロセスはブロックされていれば，準備リストに置かれる．

割振りルーチンがソフトウェア割込み保留によりプロセスにプロセッサを割当てることを回避している時，退避していたプログラム・カウンタで示される

記憶場所から再開させず，保留中の最優先順位割込み番号に関係した約束で決められている記憶場所から再開する．そのプロセスは以前の状態に戻るためにこのルーチンからスケジューラに返る．ソフトウェア割込みにマスク機能を付加することができよう．言い換えれば，その機構はスケジューラによって取扱われること以外はハードウェアでみたものと同様である．

状態情報はスケジューラが各プロセスに関して保持している情報の一部として，あるいは共用記憶領域に格納される．プロセスは相互に共用記憶領域を通して信号の授受ができる．特定のプロセスに明白な形で合図するためにも，また他のプロセスに暗黙の形で合図するためにも使えるプロセッサ間コミュニケーション設備の形式に**事象チャネル**（event channel）方式があり，これはMultics システム[36]およびCAL システム[35]において違った形で使われている．基本的な考え方は極めて単純であるが，一般性に富み，非常に強力なものである．特有な状態を持つプロセスは，システムに対して**事象チャネル**を生成するよう要求を出すことができる．システム内には多数の事象チャネルがある．許可されたプロセスのみが特定のチャネルとコミュニケーションできるようにするため，事象チャネルは保護される．事象チャネルの最も単純な形式は情報の単なる先入先出待ち行列である．その待ち行列に入れられる情報量は可変であるが，通常，1つの状態語あるいは数個の状態語に制限される．共用記憶領域は大量の情報を転送するのに利用される．

プロセスをプログラムする際にはどの事象チャネルを共同プロセスで使うかという取決めを行なう．いくつかのプロセスは共通の事象チャネネルを利用できる．あるプロセスとコミュニーケーションしたいときは，システム呼出しを通して1つあるいはいくつかの事象チャネルにアドレス付けされたメッセージを送る．システムはアドレスされたチャネルの待ち行列の最後にそのメッセージを置く．そのプロセスがコミュニケーションを求める時点に到達すると，アドレスした事象チャネル待ち行列の次の項目を要求する．システムはそれが有効であれば待ち行列から取出し，要求しているプロセスに渡す．待ち行列が空であれば，要求しているプロセスはブロックされ，メッセージを受取った時にそのプロセスを呼び起こせるように，要求プロセス名を事象チャネルに格納する．ここでの議論は最も単純な形式の事象チャネルについての概念を示している．この考えをさらに発展させたり変更することもできる．

外部割込みの処理をするルーチンもまた明白な形でスケジューラにプロセス識別タグとおそらく優先番号を送ることができる。このコミュニケーションの方法は外部機器を取扱うルーチンに要求を出す場合にもプロセス識別タグを付

ける必要がある.

暗黙の形のコミュニケーション方法は XDS-940 システムで採用されている[2]. プロセスがブロックされるとその状態ベクトル中の1語にブロックされた理由がセットされる. この語は割振りルーチンによってチェックされるアドレスおよび割振りルーチンがそのプロセスを呼び起こすための条件を示すビットなども含んでいる. ある種の動作完了時にそのプロセスを呼び起こそうとする I/O またはその他のルーチンは, どのプロセスをサービスしているのかをはっきりとは知らないが, その動作終了時にある特定の記憶場所にある値を格納すべきことを知っている. その指示は一般に単純で, 0より大であるか, 等しいか, あるいは小さいかである. 暗黙の形のコミュニケーション方法によると, ブロックされたプロセスと実行準備にあるプロセスは同一のものとなる. 割振りルーチンはリストを調べて条件が満足されているかどうかをチェックしなければならない.

明白な形のコミュニケーション方法はもっと自然な形となり, 主にハードウェア・モジュール間のコミュニケーションに使われるが, ソフトウェアで実現すると多くのコードが必要になる. 暗黙の形のコミュニケーション方法はシステム・モジュール間の結合が粗になる可能性があることから, 修正しやすいコードになる. しかしながら, どのプロセスを次に実行させるかを決める判断に余分な時間がかかる. なぜならば, 特定のスケジューリング・ルーチンが次に実行させるプロセスが見つかるまで, プロセスの各々に対応した1個以上のセルをある順序でチェックしなければならないからである. I/O 要求を含むコミュニケーションの詳細例は6.2.3節で挙げる.

5.2.5 文脈ブロック

ここでプロセッサを切換える際に保存しておく必要のある情報の性質についてもっと詳しく考察してみる. プロセッサを切換える時に保存すべき情報は1.5.2節で紹介した**文脈ブロック** (context block) である. 文脈ブロックはプログラム・カウンタ, プロセッサ中央レジスタ類の内容, アドレス・マッピングが使われていれば付加レジスタ類, プロセスの現在のアドレス空間に関するある種の表示, ならびにある I/O 機器の状態に関する表示などを含んでいる. プロセスがブロックされているか, あるいは準備リスト上にある間, この情報のうちのどの位を実際に主記憶中に残しておかなければならないのであろうか. ブロックならびに準備状態にあるプロセスが乱費するプロセッサおよび記憶リソースをできるだけ少なくしたいので, これは重要な問題である. またプ

ロセッサの休止時間を最小にするために，プロセッサの切換えに要する時間をできるだけ少なくすることも必要である．

上に挙げた情報以外に各ジョブあるいはプロセスに関係する記憶領域が通常若干あるが，その領域をそのプロセスはアクセスできず，一時的な格納，バッファ空間，定数，およびこのプロセスに固有なテーブル類のためにシステムが使う．この記憶領域は一時記憶領域（temporary storage area）と呼ばれ，XDS-940 システムでは1ページとられ，計算を始める前に主記憶内に駐在していなければならない．プロセスを実行させるために必要なすべての情報を集めたものが文脈ブロックである．

文脈ブロックはテーブル，あるいはテーブルへのポインタを含み，これは5.3.2 節で述べるプロセスの仮想空間を定めるものと類似している．この情報が補助記憶上にある場合は，システムはプロセスの記憶空間の残りの部分がどこに格納されているかを知るのに1回のアクセスが必要になる．XDS-940 のように，活動中のプロセス（実行状態，準備状態，あるいはブロック状態）が比較的少なく，各プロセスに対する記憶空間が比較的小さいシステムの場合は，いくつかの文脈ブロック情報は常に主記憶内に保持される．この方法は前もってテーブル空間を割付けることから，活動プロセス数に関して制限があるが，より少ない補助記憶への参照でプロセスの切換えが可能になる．5.3 節で述べるように，計算とスワッピングを並行して行なうために，もしプロセスあるいはプロセスの一部をいくつか主記憶内に同時に保持しようとすれば，文脈ブロックもおそらく主記憶内に入れておくべきであろう．そうでないとプロセスを続行させるために必要な情報を戻す間に休止時間が生じる．

重要な点は準備およびブロックされているプロセスについて主記憶内に必要な文脈ブロック情報は非常にわずかであること，およびプロセッサの切換え速度と主記憶の利用の間につり合い問題が存在することである．

5.2.6　準備リストの構造—割振り

準備リスト・エントリー　ここで準備リストの構造に注意を向けよう．各プロセスに関係する情報はプロセス識別タグ，優先番号，クオンタム数，呼び起こし待ちスイッチ類またはソフトウェア割込みフィールド，およびスケジューリングの骨組を決めるのに必要なその他の情報などであり，これらは準備リスト中に明白な形あるいは暗黙の形で入れられる．

XDS-940 システムで採られた方法はこのスケジューリング情報をいくつかの文脈ブロック情報と一緒に連続したセルにまとめたものである．この情報は

プロセス活用化テーブル (process-activation table) と呼ばれるエントリーのようなテーブル中にエントリーを構成する．プロセス識別タグはそのプロセスに関するテーブル・エントリーの位置によって暗黙の形で示される．この識別子は**プロセス番号** (process number) と呼ぶことができる．種々の形式の実体に対する識別子としてテーブルの位置を用いることはオペレーティング・システム設計を通して採られる一般的な方法である．

リスト処理の利用　優先権は扱いやすいように準備リスト中の位置によって取扱われる．他のシステムでも一般的であるが，XDS-940 の準備リストは主記憶中でこれらのエントリーを物理的に動かすことはせず，リンクされたリストになっている．そのようにして準備リスト中の各エントリーは次に続くエントリーへのポインタを持つ．テーブル中のエントリーには，関連したプロセスへリンクする別のポインタが保持されている．リストとしてテーブルや待ち行列を構成することはオペレーティング・システム設計で使われる一般的な技法である．最も単純な形式の準備リストを論理的に示したものを 図 5.1(a)に，またテーブルの形にしたものを 図 5.1(b)に示す．各エントリーは1つのプロセスに関する状態情報を保持するために，数語の連続した計算機語から成っている．テーブルを実現する一般的な方法は本書の残りの部分の適当なところで述べることにする．

ブロックされたプロセスは別のリストに置くことにより記録することができる．再度述べるが，もしリスト処理を利用すれば記憶装置内での移動は必要ない．あるプロセスの優先順位を変更したり準備リストに新しいプロセスを加えるためには単にリストに新しい要素を挿入すればできる．挿入する時は，影響を受けるエントリーに関連したポインタ語に格納されている値を変更するだけである．

優先順位の割当て　単純な**先着順サービス方式**による優先順位およびクオンタム数の割当については先に述べた．この方式はプロセスを実行状態から取り除いたり，呼び起こす際に，それを準備リストの底に置く．各プロセスはブロックされなければ，ある一定のクオンタムの間実行する．このスケジューリング方法はしばしば**ラウンド・ロビン** (round robin) **・スケジューリング**と呼ばれる．

もう1つの一般的な方法はある種の多重レベル優先順位方式と可変長クオンタムを使うものである．一般に使われる2つの変形について述べるが，別の変

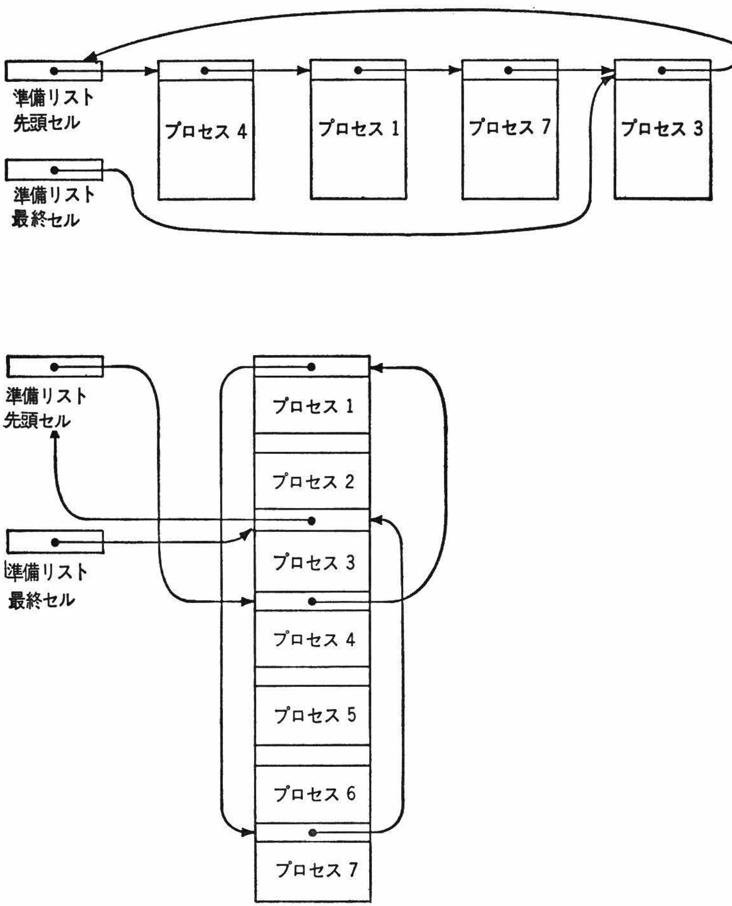


図 5.1　簡単なリスト構造をした準備リスト
(a) 論理的構成；(b) テーブル構成

形も可能である．先に論じた骨組の中でさらに複雑な方式も考えられる．

多重レベル方式を使う背景にある動機はシステムを交信中のユーザに対して応答良くさせることにある．多重レベル方式の一般的な特徴は計算（CPU）向きのプロセスには低い優先順位を与え，一方会話向きのプロセスには高い優先順位を与えることである．多重レベルの方法は準備リストを違った優先順位の部分に分ける．ある優先順位レベルに加えるプロセスは，通常そのレベルの最後に入れる．1つのレベルの中でのサービスは先着順サービスである．したが

って，プロセスを実行させるために選び出す時点で，割振りアルゴリズムは常にまず最優先順位の区分から捜し始める．低い優先順位レベルにあるプロセスはそれより高い優先順位レベルに準備状態のプロセスがない時だけ実行される．たとえば，レベル2のプロセスはレベル1のすべてのプロセスが実行されてしまうまで実行させてもらえず，たとえプロセスが引き続きレベル1に加えられてもそうである．この方式に以下に述べるように付加的な特徴を加えない限り低いレベルにあるいくつかのプロセスは実行されないか，あるいは応答を受取れない程まれにしか実行させてもらえない可能性がある．

XDS-940の多重レベル方式　XDS-940システムに見られる多重レベル方式をここでは多少簡単にして議論する[2]．優先順位の順にテレタイプ，I/O，短時間クオンタム，および長時間クオンタムと呼ばれる4つのレベルがある．このような多重レベル・リストをリスト処理で実現したものを 図 5.2 に示す．テレタイプに字を要求してその字が得られないか，あるいは出力バッファが一杯でブロックされていたプロセスが呼び起こされると，テレタイプ・レベルへ入れられる．同様にその他の I/O サービスのために種々の理用によりブロックされていたプロセスは，呼び起こされると I/O レベルに入れられる．

システムは**短時間クオンタム** (short quantum) と**長時間クオンタム** (long quantum) という2つの長さのクオンタムを用意している．プロセスは常に短時間クオンタムで実行を許され，この時間の終りに他に準備状態のプロセスがなければ続行できる．短時間クオンタムの目的は，プロセスのスワップ・インに対する費用が高くつかないようにするため，計算を有効に行なうことにある．またこの方式は高い優先順位のプロセスが短時間クオンタム中または終了時に出て来た時にプロセッサを先取りできるようにしている．あるプロセスがそれより高い優先順位のプロセスが準備状態になったために短時間クオンタム後に追い出される時，そのプロセスは短時間クオンタムに入れられる．プロセスの実行中，このプロセスの長時間クオンタムに関連して計算機に格納されている数が各クロック・パルス毎に減らされる．そのプロセスがブロックされると長時間クオンタム値は保持され，次のそのプロセスの実行時から続けて減らされる．このようにして結局はその長時間クオンタム数は0になり，この場合，そのプロセスは低い優先順位レベルで長時間クオンタム・レベルに移される．この方法はすべてのプロセスが適切な応答性で実行できるようにする．この方式はプロセスが準備リストの高い優先順位レベルになる回数を制限するように工夫したものの1つである．

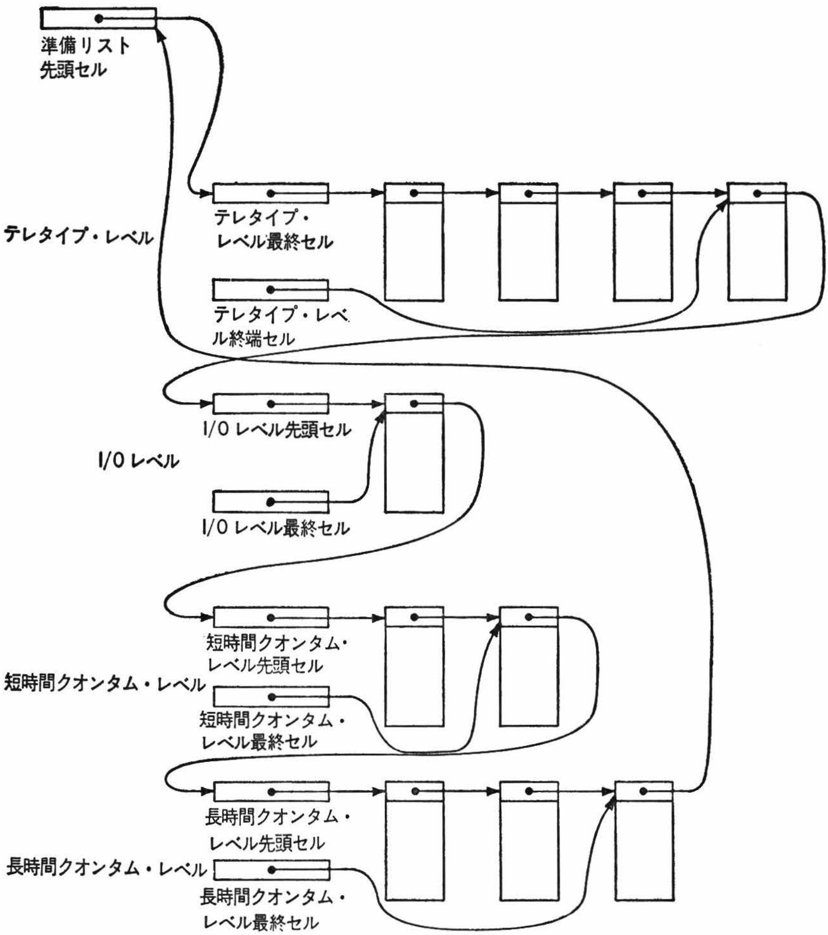


図 5.2　リスト処理で実現した多重レベルの準備リスト

XDS-940 スケジューリング・アルゴリズムにおけ1つの工夫点を述べる必要があろう．同じ技法が他のシステムでも使われている．あるプロセスが I/O 要求を出すと，直ちにそれをブロックせず，要求された I/O サービスが終了するまでの予想時間を含んだテーブルを調べる．もしこの時間が残りのクオンタム量よりも少ない時は システムはそのプロセスに I/O 要求の処理を行なわせ，プロセスをブロックせずに待機する．この動機はもし I/O 動作が終了できるならば，そのプロセスをスワップ・インするための費用を節約できるからで

ある．そのプロセスをブロックしたとすると，おそらく直ちに呼び起こすにもかかわらず，スワップ・アウトし，そしてまたスワップ・インすることになろう．

実行時間による多重レベル優先順位の構成　多重レベル方法のもう1つの変形はいくつかのレベルの各々に異なったクオンタム長を割当てるものである[4,25]．ジョブを実行する時は，以前にブロックされていなければ，それが入っていた待ち行列のクオンタム長で指定される時間だけ実行できる．たとえば，なんらかの基準にしたがって1つのレベルのクオンタム長を調整した方がよい．すなわち，5.2.3 節で述べたように，そのレベルに達したプロセスの大部分が選択したクオンタム期間に終了できるようにする．プロセスは生成されると最上レベルの最後に置かれる．プロセスはそのレベルに対して指定された時間を使い果たすと，次に低いレベルの最後に置かれる．このようにして，計算(CPU) 向きのジョブは次第に下位のレベルに下げられていく．ジョブがテレタイプあるいはその他の I/O のためにブロックされ，その後呼び起こされると，それは再び最上レベルの終りに置かれる．クオンタム長の設定はシステム特性の関数であり，実際のシステムでの実験あるいはシミュレーションによってのみ決定できる．クオンタム長は重要なパラメータであり，システムの応答性や利用率に鋭敏に影響するので注意深く考慮する必要がある[4,21]．

記憶装置やその他のリソースの最大利用率を考慮すると，スケジューリング・アルゴリズムはさらに複雑になる[24]．5.3 節で行なった記憶割付けの議論の中で効率的なスケジューリング・アルゴリズムはおそらくスワッピングや記憶割付けのために選択するアルゴリズムとは無関係に設計できないだろうということを示した．すべてのアルゴリズムはおそらくハードウェアの性能特性および特定の設置でのユーザ社会のある範囲の特性を反映するように選択すべきであろう．

ま と め　スケジューリングは2つ時点で行なうことができる．すなわち，プロセスが準備リストに置かれたとき，およびプロセスがプロセッサに割振られるときである．実用的なスケジューリングを行うにはそのための計算をあまり多くすべきではない．一般的な方法はプロセスを準備リストに入れる際にスケジューリングを行なう．準備リストは優先順位レベルを取扱うように構成でき，またプロセスに優先順位を割当てる方法はたくさんある．大部分の方法は会話向きプロセスに高い優先順位を与えるようにしているが，最良の方法

であると一般的に同意を得られているものはない．割り振りは準備リスト上の最優先順位のプロセスを単純に選び出すという問題になる．

5.2.7　システム機能のスケジューリング—割込みルーチン

いままでのスケジューリングの話は主にユーザ・プロセスに対して適用される．スケジューラはまたシステム・プロセスをも制御しなければならない．多くのシステム・ルーチンはユーザ・プロセスのように仮想プロセッサ上では実行しない．その結果，多くのシステム・ルーチンはユーザ・プロセスとは異なった特性を持つ．以下で行なう割込み処理ルーチンの議論でこれらの相違点を指摘する．システム・ルーチンは割込み，ユーザ・プロセスによるサービス要求，およびI/O機器からの状態情報によりブロックされたりブロックを解かれたりするものと考えられる．多くのシステム活動はスワッピングの必要がない常駐ルーチンによって行なわれる．割込みあるいはユーザの要求が到着すると，制御は適当なルーチンに移され，要求された動作が行なわれて制御はそのユーザ・プロセスに戻される．システムはI/O要求ならびに一時に1つのプロセスだけしか利用できない機器に関するその他のサービスを取扱うために，自分自身の準備リストを持つ．I/Oのスケジューリングについては第6章で議論し，これらの考えに対する詳細例を6.2.3節に挙げる．

システム・ルーチンの途中でクオンタムがなくなったとき，システム・ルーチンのスケジューリングに関連した問題が出てくる．もしそのシステム・ルーチンが割込み処理ルーチンだとすると，スケジューラに入る前にそれを終了させる必要がある．しかしながら，割込みルーチンから割込まれたユーザ・プロセスではなくスケジューラへ戻るためのなんらかの機構がなければならない．もしシステム・ルーチンがサービスに対するユーザ・プロセスからの直接要求を扱っており，その途中でクオンタムを使い果してしまったときは，システム設計によって異なるが，2つの一般的な動作をとり得る．システム・ルーチンがリエントラントでないか，あるいは実行が終了していないときにシステムを危険な状態にしておいてもよければ，そのルーチンが終了するまでクオンタム終了信号を無視すればよい．この場合，ユーザ・プロセスではなくスケジューラに戻らなければならない．再び述べるが，スケジューラに戻るためのある種の機構を用意する必要がある．このような機構について次に述べよう．システム・ルーチンがリエントラントで終了まで待つ必要がなければ，クオンタム終了時に直接スケジューラに入ることができる．

システム・ルーチンの処理中にクオンタムが終了した時に，システム・ルー

チンからユーザ•プロセスではなくスケジューラに制御を移すために XDS-940 システムで使われているハードウェア機構はユーザ・モード・トラップ (user mode trap) と呼ばれる. それは次のように動作する. クロック割込みルーチンがクオンタム終了が起こったということを知り, しかもシステム・ルーチンが現在実行中である場合は, ユーザ・モード・トラップがセットされ, スケジューラではなくそのユーザ・ルーチンに制御が戻される. その後システム・ルーチンが終了し, システム・モードからユーザ・モードに戻ろうとした時にトラップ信号が発生されてシステムに制御が戻される.

4.2 節の割込みについての議論の中で, 現在の割込みシステムは割込みルーチンをシステムの通常のプロセス・スケジューリングおよび優先順位割当て機能の外で取扱う必要があるということを指摘した. システム中の他のプロセスと違って, 割込みルーチンは仮想プロセッサ上では実行しない. これらは基本システムの一部としてハードウェア・プロセッサ上で実行する. その結果, 仮想プロセッサに有効なシステム機能および機構, すなわち保護, ファイル・システムへのアクセス, 仮想プロセッサ間のコミュニケーション, ユーザの端末へのアクセス, およびユーティリティ・プロセスへのアクセス, は割込みルーチンには容易に利用できない. 一方, 仮想プロセッサ上で実行しているプロセスは割込み機構を通してシステムに入ってくる信号を処理することは容易には許されない. 実際には, スケジューリング機能は 2 つの部分に分けられる. 1 つはハードウェア割込み機構を通してシステムに入ってくる信号を取扱う部分であり, 他方は仮想プロセッサによって発生される信号を取扱う部分である. ハードウェア信号に割当てられる優先順位は一般にそれらの間で固定されており, 割込み禁止でなければハードウェア割込み処理ルーチンは常に仮想プロセッサ上で実行されるプロセスよりも高い優先順位が与えられる. これら 2 つのスケジューリング機能部分のインタフェースを取り持つ機構を開発できる. 割込みルーチンとその他のプロセスを二分して扱うスケジューリング・システムは動作させることはできるが, 機構を最小にし機能的に関係したタスクを統一して扱うという望ましい目標に反する.

割込みルーチンを仮想プロセッサ上では実行させない理由は 1 つの仮想プロセッサから他の仮想プロセッサへハードウェア・プロセッサを切換えることが時間のかかるタスクであるからである. 概念的に含まれることは, 状態ベクトルの切換えである. 状態ベクトルは計算機の中央レジスタ類, 仮想プロセッサの記憶マップ, クオンタム長, あるいはシステム・ルーチンのための一時記憶領域, I/O 機器およびその他の状態情報などから成っている. 現在の割込みシ

ステムの長所は退避する状態情報を最小にしてハードウェア・プロセッサを割込み処理ルーチンに切換えることができることである．割込み処理ルーチンは種々の中央レジスタ類のような壊わす可能性のあるいかなる状態情報も退避する責任を負う．

したがって，もしハードウェアおよびソフトウェアによる呼び起こし信号を統一的な方法で取扱おうとすれば，必要な時に早急に仮想プロセッサの状態ベクトルを切換える機構を開発することが要求される．スケジューリングを行ないプロセッサ切換えの手助けをする特殊なマイクロプログラム制御プロセッサを実現するという Lampson の提案[2] について 4.2.4 節で議論した．この特殊プロセッサは記憶装置にアクセスし，準備リストを操作し，プロセスの割振りやその他のスケジューリングを行なう．前に述べたことは主記憶内に常駐しているシステム・ルーチンも含めている．

通常，主記憶内にないシステム・ルーチンが必要になると，主記憶にスワップ・インする．時間が重要な要因になっていなければ，このシステム・プロセスに対する状態ベクトルを設定し，このプロセスを準備リストに置く．Multics および XDS-940 の双方とも低い優先順位の検査あるいはその他の操作はシステムが生成するプロセスで行なう．これらのプロセスの各々に対する状態ベクトルを生成し，それを準備リストに置く．このようなプロセスの一例に数秒毎に呼び起こされてシステム・テーブル類が混乱していないことを検査したり，I/O 機器あるいはチャネルの動作が妨害されていないかを調べるものがある．XDS-940 システムでは非常駐システムの大部分はすべてのユーザに割当てられるプロセスと考えられる．そのプロセスは主記憶内にたった1つの複写があればよいようにリエントラント・コードで書かれる．

Multics システムでは，システム・セグメントはユーザのプロセスの一部として割当てられる．このシステム・セグメントはリエントラントである．一部は常駐しているが，非常駐のものある．

5.2.8　インタロック

もう1つ簡単に述べる必要のある話題は**インタロック**[2,5,32] (interlocking) の問題であり，これには多くの見方がある．まず一般の状況を考察してみよう．いくつかのプロセッサ(ハードウェアあるいはソフトウェア)が共通のデータ・ベースを共用する場合，ある種の注意が必要になる．1つのプロセスによってそのデータ・ベースが変更されようとしている時に他のプロセスにそれを読んだり書いたりすることを許すのは危険である．いくつかのプロセスは危険を犯

さずに同時に共通データ・ベースを読むことができる．それゆえ，データ・ベースを変更しようとするプロセスのために，それをアクセスしようとするその他のプロセスを**ロック**（lock）する必要がある．このようなロックを持つことはデータ・ベースへのすべてのアクセスで使われるルーチンが1つあることを示唆している．

ルーチンが共通データ・ベースにアクセスしようとする時はまずロックを調べる．もしロックがセットされていればそのルーチンは待たなければならない．もしロックがセットされていなければ，ルーチンはロックをセットしてからデータ・ベースをアクセスし変更する．変更が終った後でロックは解除される．ロックをチェックし，それをセットするまでの間に他のプロセスがまたロックをチェックできないようにロック機構を実現することが重要である．この状況が生じれば，1つのプロセスはデータ・ベースのアクセスを同時に得られることになり，それはまさしくロックが阻止すべき状況である．

種々のハードウェアおよびソフトウェア機構がロックを実現するために利用できる[2,15,32]．Lampson は次のような機械語命令を用意することを提案している[2]：

1. アドレスされた記憶語の内容をテストする．
2. もしそれが負であればスキップする．
3. もしそれが正であればアドレスされた語を負にセットしてスキップしない．

ロックのテストおよびセットのための命令を TSL と呼ぶことにすると，次の命令シークエンスはインタロックを実現する：

```
TEST    TSL       LOCK
        BRANCH    DATA BASE
        BRANCH    TEST
```

もし LOCK の内容が負であれば他のプロセッサに対してそのデータ・ベースはロックされる．LOCK の内容が正になった時にデータ・ベースのロックは解除され，プロセッサはアクセスを得ることができる．そして他のプロセッサが同時にそのデータ・ベースに到達することがないように同じ命令を使ってロックをセットできる．Lampson によって指摘された上の方式には1つの欠点がある．それは締出されたプロセッサは引き続きロックをテストして記憶サイクルを浪費することである．この問題を回避するためにはデータ・ベースから

締出されたプロセスはロックが解除された時に呼び起こされるリストに自身自身を置いてブロックすればよい．現在そのデータ・ベースをアクセスしているルーチンがロックを解除して，そのデータ・ベースのアクセスを待っている次のプロセスを呼び起こすような標準ルーチンを通って出てくればよい．

Lampson はもう1つ別のインタロック機構を提案した[2]．それはある一定時間の間だけあるブロックのコードから他のプロセッサを完全に締出すもので，ロックをチェックするために記憶サイクルを使ったり，ロックが解除されるまでプロセスをブロックするソフトウェア機構は必要としない．その考え方は彼が PROTECT と呼んでいる特殊命令で実現される．プロセッサがその命令を実行すると，一定時間その命令は再び実行できない．もしその他のプロセッサがこの命令を実行しようとすると，待ち時間が過ぎるまで一時的に休止状態になる．

これを実現する一方法は，PROTECT 命令の実行でカウンタをセットすることである．そのカウンタは自走モードで一定の速度で0に減らされる．そのカウンタが0より大きい時に他のプロセッサが PROTECT 命令を実行しようとすると，そのプロセッサはカウンタが0になるまで停止させられる．0になった時点で待たされていたプロセッサは PROTECT 命令を実行し処理を続ける．多重プロセッサ・システムでは数台のプロセッサが PROTECT カウンタが0になるまで待って停止している場合を処理できるように，PROTECT 命令にアクセスする優先順位を割当てなければならない．上で述べた機構はシステム・テーブルを変更したり待ち行列エントリーの位置を変更するために数命令が実行されている間，プロセッサを締出すのに非常に有効である．しかしながら，十分な保護を行なうためには1つ特徴を付け加える必要がある．すなわち，割込みシステムの信号は PROTECT カウンタが0でない間は自動的に無視されるようにすることである．

この機構は PROTECT 動作を行なっている間に実行中のコードが記憶装置にないページにアクセスし，**ページ・フオルト**（page-fault）**割り込み**が発生する時に問題を生じる．この割込みは無視できない．Lampson は PROTECT 動作中に発生するすべてのアドレスを同一ページに入れるか，主記憶内にあることを保証したページに入れるか，あるいは問題とする動作が行なわれる前に参照する別のページに入れるようにすればこの問題は処理できると提案した．最後の場合はページ・フォールトからの戻りで PROTECT 動作を実行しても何ら困まることはない．

他にもインタロックの問題がある．2つのプロセスAおよびBが1台のプロ

セッサを共用している場合，次のような状況が起こり得る．プロセスAはロックをセットしてデータ・ベースにアクセスし，その後ブロックされる．プロセスBはロックをテストするループに入いる．この時点で計算機はプロセスAが計算機の制御を再び得ないと**デッド・ロック**（dead lock）に落ち入ってしまう．タイムシェアリング・システムではプロセスBは結局そのクオンタムを使い果たし，プロセスAは再び計算機の制御を得られることになろう．

2つのプロセスは別の方法でも互いにデッドロックを引き起こす．プロセスAがリソースXを使用し，プロセスBがリソースYを使用しているとする．プロセスAがXを解放せずにリソースYへのアクセスを要求し，ブロックしてYが空くのを待つ．その後，プロセスBがYを解放せずにリソースXへのアクセスを要求し，ブロックしてXが空くのを待つ．この時点で双方のプロセスはシステムがリソースXあるいはYを解放できない限りデッドロックになる．もしプロセスが一時に1つのリソースしかアクセスできないようにすればこの状況を避けることができる．プロセスが一時に1つ以上のリソースの制御を得られるようにするには，設計は潜在的なデッドロック状況を認識する手段を用意しなければならない．色々な形で現われるインタロックの問題は設計者が気が付かなければならない厄介な問題である．

5.2.9 まとめ

本節で多くの重要な概念を招介した．プロセスは3つの状態のうちの1つにあることを知った．すなわち，ハードウェア・プロセッサ上で実行している状態，なんらかの条件が満足されるまで待ってブロックされている状態，あるいは実行する準備ができている状態である．実行準備状態にあるプロセスは一般に優先順位によって示される準備リスト上に保持される．プロセスは優先順位にしたがって準備リストからプロセッサに割振られる．準備リストは応答性やその他の要求を満足するためにいくつかの優先順位レベルで構成する．その他の多くのスケジューリング機構をここで示した骨組に利用できる．

タイマおよびクオンタム数を使って，プロセスには長時間の実行を行なわせない．通常，準備リスト上にある高い優先順位のプロセスが実行中のプロセスからプロセッサを先取りするような手段もまた用意される．

プロセスとシステム・ルーチンが相互コミュニケーションする方法を用意する必要がある．明白な形および暗黙の形のコミュニケーション機構について述べた．ソフトウェア割込みの概念はプロセス相互のコミュニケーションのための1つの機構であり，事象チャネルの概念はもう一方のものである．現在のシ

ステムではハードウェアおよびソフトウェアのブロックならびに呼び起こし信号の取扱いに統一性が欠けていることを明らかにした．

常駐および非常駐のシステム・ルーチンについての考え方を紹介した．常駐ルーチンは頻繁に生ずる要求あるいは高い優先順位の要求をサービスするために常に主記憶内に駐在する．非常駐ルーチンは頻度の低い要求あるいは低い優先順位の要求に使われ，補助記憶内に駐在することができる．

プロセスおよびプロセッサのインタロックの潜在的に面倒な問題があることを示した．共通のデータ・ベースを処理する場合，1つのプロセスを他のプロセスから適切に分離させるためにインタロックは注意深く設計されなければならない．あらかじめ考慮しておかないとあるプロセスが他のプロセスの条件によってどちらも先へ進めないようなデッドロックの状況に落ち入ることがあることを示した．

最後にプロセッサ・スケジューリングの設計はおそらくそのシステム内のリソース割付け問題を切り離しては考えられないだろうということを指摘した．

5.3　記憶の割付け

5.3.1　概　　要

記憶割付け問題は2つの主要な部分に分けてみることができる．すなわち，(1) システム内にある各プロセスの仮想記憶空間が主記憶内か，補助記憶上か，あるいは両方か，およびどこにあるのかといったことを記録する記録保持機能，ならびに主記憶および補助記憶のどの領域が空いているかを記録する記録保持機能であり，そして，(2) ある時間に1つ以上のプロセスが主記憶中にある時に必要となり，実行しようとする新しいプロセスの空間を作るためにどのプロセスあるいはプロセスの部分を補助記憶上に移すかを決定する主記憶割付け(スワッピング)，判断，およびその方法である．補助記憶上の未使用空間を記録する問題は6.1.3節で論じる．Multics のようなシステムの仮想記憶空間を記録する問題は第2章で若干詳しく述べた．XDS-940の 記録保持機能はいくつかの基本的問題を示しているのでここでさらに詳しく述べる．この議論の後で記憶割付けおよびスワッピンクの問題についてさらに詳しく調べることにする．

5.3.2 記憶空間テーブル

概　要　2組のテーブルが必要になる．一方のテーブルはプロセスの仮想空間を記録し，他方は実記憶の使用状況を記録する．これらのテーブルの間はリンクがとられていなければならない．次の例を理解するために，読者はXDS-940のページング機構と2.4.2節で述べたリラベリング（マップ）レジスタについて熟知している必要がある．

実記憶の記録　プロセスの実行中，そのプロセスの論理ページが格納されている主記憶内の実際の物理ブロックは実リラベリング・レジスタ，またはマップによって示される．入ってくるプロセスに割当てられる主記憶内の物理ブロックを決めるために，記憶割付けルーチンは2つのテーブルを使う．1つのテーブルは**実記憶テーブル**（real-memory table）（RMT）と呼ばれ，主記憶の種々の物理ブロックをどのプロセスのどの論理ページが使っているかという情報およびどのブロックが未使用で，プロセスのページに使われていないかという情報を含む．RMTは物理ブロック番号によって順序付けられている．もう一方のテーブルは**実記憶カウンタ**（real-memory counter）（RMC）と呼ばれ，物理ブロック内に格納されているコードを使っているプロセスの番号に関する情報を含む．たとえば，リエントラントのテキスト編集システムは数人のユーザで使われるが，本当に必要でないとき以外はスワップされないようにすれば効率が良くなる．入ってくるプロセスにどのブロックを選択して割当てるかを決定するのを助けるために，物理ブロックが使われた頻度を示すカウンタが付加される．この種のカウンタについては5.3.3節で述べた．上で述べたテーブルの各々は主記憶の各物理ブロックに対して1つのエントリーを持つ．

仮想記憶の記録　第2章で仮想記憶はセグメント・ディスクリプタ・テーブル，およびページ・ディスクリプタ・テーブル内に記録されることを知った．XDS-940では，多くの場合1つのジョブはいくつかのプロセスから成るが，そのジョブに関連した実ページあるいは記憶空間は**プログラム記憶テーブル**（program memory table）（PMT）[22]と呼ばれるテーブル内に記録される．したがって，各ジョブに対して1つのPMTがあることになる．PMT（ページ・ディスクリプタ）の中には各ユーザの論理ページに対して1語の記憶領域がある．プロセスはページを共用でき，このような場合はエントリーを重複するので，ジョブ内の各プロセス毎に別々のページ・テーブルを持つことはな

い．しかしながら，これから述べようとする機構により別々のページ・テーブルがあるような論理的な効果が生み出される．この方式への動機づけは次節でさらに論議する．PMT 内の語（ディスクリプタ）はそのページが物理記憶（主記憶あるいは補助記憶）のどこにあるかを示す．そのページが主記憶内にある時にはこの PMT 語はページが格納されている先頭物理アドレス（物理ブロック）を示し，ページが補助記憶内に格納されている時はその物理アドレスがそこに入れられている．

PMT エントリーは特に順序付けられてはいない．すなわち，1つのジョブに属するすべてのページは連続した PMT セル内にあるが，PMT 内のエントリー位置はプロセス内のページの論理的順序に対応させる必要はない．論理的順序はそのプロセスの状態ベクトルの2語に格納される．これらの2語は2つの実リラベリング・レジスタと同一形式になっており，**擬似リラベリング・レジスタ**(pseudo relabeling register)と呼ばれる．各プロセスに対して一組の擬似リラベリング・レジスタがある．実リラベリング・レジスタと擬似リラベリング・レジスタの違いは擬似レジスタのマップ・バイトは実記憶ではなく PMT エントリーを指していることである．擬似リラベリング・レジスタ内のマップ・バイトの順序はプログラムのページの論理的な順序となっている．すなわち，バイト1は PMT の論理ページ1に関しての情報が格納されているエントリーを指しており，以下も同様である．擬似リラベリング・レジスタのバイトおよびそれらが指している PMT 内の適当なエントリーは，各プロセスに対して別のページ・テーブルがあるような効果をもたらしている．

RMT は実際には明白な形の情報は含まないが，PMT へのポインタを含んでいる．RMT 内のポインタと PMT エントリー内に含まれる物理記憶のページ・アドレスは実記憶および仮想記憶間の関係になっている（このことについては本節でさらに議論する)．記憶割付けルーチンがある物理ブロックを入ってくるプロセスに割当てることを決めると，そのブロック内のあらゆる値の情報を補助記憶に移さなければならない．RMT を使うことによってシステムは適切な PMT エントリー内のアドレスをそのページが移される補助記憶の記憶場所を示すようにセットしなおすことができる．この機構を 図 5.3 に示す．システムを呼出すことにより，プロセスは擬似リラベリング・レジスタ，あるいは PMT エントリーの内容を変えることができ，このようにしてオーバーレイが行なわれる．

要約すると，あるプロセスがブロックされ他のプロセスを実行しようとする時に XDS-940 では次のようなことが行なわれる．他のシステムでも同じよう

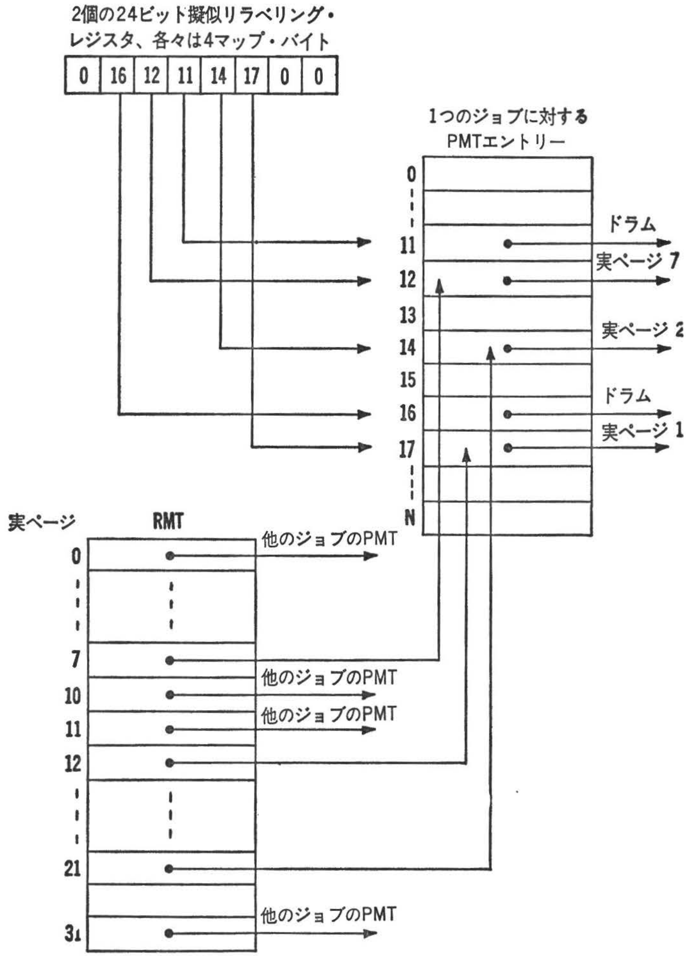


図 5.3 XDS-940 における記憶割付けテーブルの関係

なことになるであろう：

1. 擬似リラベリング・レジスタおよびPMTを使ってチェックが行なわれ，実行しようとするプロセスが既に主記憶内にあるかどうか，もしなければどの位の物理ブロックが必要かが決められる．もし新しいプロセスが既に主記憶内にあれば，直ちに制御が移される．

2. RMT，RMC，および利用カウンタを使って，記憶割付けルーチンはどのブロックが空になるかを判断する．（この点はさらに5.3.3節で述べる）
3. 擬似リラベリング・レジスタおよび PMT を使って，主記憶に持ち込まれるページの記憶場所と論理的順序が決められる．
4. ページを記憶装置に入れたり出したりするための I/O のコマンドが用意され，ページのスワップが行なわれる．
5. 実リラベリング・レジスタが設定される．
6. 計算機レジスタ類に状態ベクトル内に格納されている値が復帰され，プロセスが開始される．補助記憶上の空間を割付ける方法は重要な概念であり，6.1.3 節で議論する．

実記憶と仮想記憶の関係　ページングを用いたシステムにおいて，プロセスに関するページを記録する概念的に最も簡単な方法は，各プロセスに対するページ・テーブルを使うことである．もしプロセスがあるページを共用するときは，各ページ・テーブルに共用されるページに対するエントリーが別々に存在することになる．ページ・テーブルを常に主記憶内に保持していない場合でも，必要となる余分な空間は重大な問題にはならない．

XDS-940 の設計者達は1つのプロセスから他のプロセスへプロセッサの切換えを高速に行うために仮想記憶情報を主記憶内に保持しておくことに決めた．このことが擬似リラベリング・レジスタや PMT 機構を用いる1つの動機である．この機構を使うのにさらに別の動機がある．ページ・テーブルが使われるものとする．プロセスの各ページ・テーブルには共用しているページのエントリーが別々にあることになる．共用されるページを移動するとき，そのページに対するディスクリプタを含んでいる各ページ・テーブルを更新しなければならない．実記憶テーブル内のエントリーはどこを指し示せば良いだろうか．このエントリーは連続したブロックまたはリストでつながった語を指している．それぞれの語はこのページを使っているプロセスのページ・テーブル内の適当な位置を指している．そのページを主記憶から取除く際に，そのブロック内の情報がどのページ・テーブル・エントリーを更新するかを示すのに使われる．このような雑用は XDS-940 で採用されている方法を使うことにより不要になる．ページを動かすときには1つの PMT エントリーを変更する必要が生じるだけである．

この機構は1つのジョブ内のいくつかのプロセスが1つのページを共用する

場合を含んでいる．いくつかのジョブのプロセスがページを共用する場合についてさらに述べる必要があろう．上で述べたPMTによる方法では1つのページを共用する各々のジョブはそのPMT内に共用するページに対するエントリーを必要とする．これは余分な空間を必要とし，共用されるページを動かす時にはいくつかのエントリーを更新する問題がまだ残っている．

XDS-940 システムではこの問題を**共用記憶テーブル** (shared-memory table) (SMT) を呼ばれるテーブルで解決しており，このテーブルをジョブの共用可能なページを記録するのに使っている．XDS-940 におけるこのような共用可能なページの例としてはコンパイラ，テキスト・エディタ，およびその他のサブシステムのページがある．共用されるページを表わす擬似リラベリング・レジスタ内のエントリーはPMTエントリーではなく適切なSMTエントリーを指すようにコード化される．

セグメンテーションを行なうシステムでは，共用されるセグメントはただ1つのページ・テーブルのみを持ち，その中にはページを動かすときに変更が必要となる情報のみが含まれる．しかしながら，もしページ・テーブル自体を動かすときは共用セグメントを使うすべてのプロセスに対するセグメント・ディスクリプタを適当に変更する必要がある．

セグメンテーションおよび厳密なページ化システムの双方に有用で，いくつかのプロセスに使われ，変更される情報の複写を保持するためのもう1つの方式に**ハッシュ・テーブル** (hash table) あるいは**連想テーブル** (association table)[26] がある．ページ化したシステムの場合について述べることにする．各プロセスに対して1つのページ・テーブルがある．ページがインデックスとしてページ番号を用いてアドレス付けされていれば，ページ・テーブルはプロセスのアドレス空間内の各ページに対して1つのエントリーを持つことになる．ページ・テーブルはまたページ番号でアドレスされる擬似的な連想記憶すなわち内容でアドレスする (content-addressed) 記憶として（以下に示すような方法で）構成し，アドレスするようにできる．このような構成で，ページ・テーブルはアドレス空間の各ページに対してではなく使われる各論理ページに対して1つのエントリーを持てるようにする．以下の議論を簡単にするために，ここでは前者の構成を使っているものとする．述べようとする方式では，ページ・テーブルのエントリーはそのページが通常駐在している補助記憶アドレスと保護情報のみを含み，主記憶の記憶場所は含んでいない．

ページ・テーブルは主記憶アドレスを持たない．記憶内容はハッシュ・テーブルあるいは連想テーブルと呼ばれる別のテーブル内に記録される．ハッシュ

・テーブルの各エントリーは補助記憶アドレスと主記憶内に入っているページに対する実記憶アドレスを含んでいる．5.3.2 節で議論した実記憶テーブルのように，ハッシュ・テーブルのエントリーは実記憶ブロック番号によって順序付けられてはいない．そのかわりにそのエントリー位置は**ハッシュ・コード化**（hash-coding）**アルゴリズム**によって決められる．

内容でアドレスする記憶を模擬するために使うテーブル検索法の一方法がハッシュ・コード化である．内容でアドレスする記憶の概念は第 2 章で述べたが，そこではハードウェア・マップとして使っていた．内容アドレス記憶内の語あるいはレコードは数フィールドから成っている．1つあるいはそれ以上のこれらのフィールドの内容はキーとして，そのキーを含むすべての語あるいはレコードを検索するのに使うことができる．

ハッシュ・コード化では，テーブルをアクセスするのに使うキーに対して変形（ハッシュ操作）を行なう．その結果はキーを含むレコードを入れるテーブルのアドレスになる．検索の際にキーをハッシュ操作すればそのキーを含むレコードが格納されているアドレスが得られる．いくつかのキーをハッシュ操作すれば同一アドレスになる可能性があり，この場合衝突（collision）が生じたという．衝突を取扱うために多くの方法がある．1つの簡単な方法は次のようなものである．もし記憶場所が既にエントリーを持っていると，この記憶場所の下にある次の空場所を新しいエントリーを格納するために使う．そこで検索の際はハッシュ化されたキーに対応した記憶場所にあるキーに対して使われているフィールドの内容をアクセス・キーと比較する．もしそれらが一致すれば，そのエントリーは所望の情報を含んでいることになる．もし一致しなければ，ハッシュ操作によって得られたアドレス以下のエントリーに対して所望のキーを含むエントリーが見つかるまで線形検索を行なう．ハッシュ・テーブルは通常循環状になっていると考えられる．すなわち，先頭のエントリーは論理的に最終エントリーに続いている．ハッシュ・テーブルの大きさおよびハッシュ・アルゴリズムは2つの補助記憶アドレスがハッシュ操作によって同一ハッシュ・テーブル・アドレスになる確率を決定する要因となる．もしハッシュ操作アルゴリズムがランダムなアドレスを作り出すとすれば，衝突エントリーの配置は仮にテーブルが75%詰っていても，衝突エントリ間の距離は2を下まわる程度である．もし衝突が生じなければ明らかに検索は早くなる．テーブルの大きさおよびハッシュ操作アルゴリズムの選択についてはより詳しく文献[28,29]で議論されている．ハッシュ・テーブルを使った一方法を簡単に紹介したところで，主記憶内のページを記録するためにどのような考え方が使われるかをみる

ことにしよう．

主記憶内に入れるページの補助記憶アドレスは，ハッシュ・テーブル中のエントリーを決定するために，ハッシュ操作が施される．作り出されたアドレスにエントリーがあれば，この点から下にある最初の空エントリーを使う．ハッシュ操作により決定した記憶場所あるいはこの点を越えた最初の空場所にページの主記憶アドレス，補助記憶アドレス，およびこのページに許されたアクセス権を含むエントリーを作る．

主記憶からページを取除く時はそのページと同一の主記憶アドレスを持ったエントリーが見つかるまでハッシュ・テーブルを検索する．このエントリーにはそのページを格納する補助記憶アドレスが収容されている．そのページの補助記憶アドレスを早く得るためにハッシュ・テーブル内を指すポインタと共に実記憶テーブルを使う．その後スワップ・アウトされるページのエントリーをハッシュ・テーブルから取除く．

プロセスを主記憶内に持込むときに，システムはそのプロセスのどのページが主記憶内に既に入っているかを調べなければならない．このチェックは以下のように行なわれる．各ページの補助記憶アドレスにハッシュ操作を施す．そして，ハッシュ操作で得られたアドレスからハッシュ・テーブルを調べ始める．ハッシュ・テーブル・エントリー内の補助記憶アドレスと主記憶に持込むページの補助記憶アドレスを比較する．それらが一致すればそのページは既に主記憶内にあり，そのハッシェ・テーブル・エントリー内にある主記憶アドレスはページがある記憶場所を示している．補助記憶アドレスが一致しなければさらにハッシュ・テーブルを調べる．そのページに対するエントリーがハッシュ・テーブル中になければ，ハッシュ・テーブルを終りまで線形検索することになる．取扱っているページの補助記憶アドレスと一致するエントリーがハッシュ・テーブル中になければ，そのページを補助記憶から主記憶内に持込み，ハッシュ・テーブル内にそのためのエントリーを作る．

この機構には次のような望ましい特長を持っている．すなわち，

1. システム中の各ページはそれが主記憶内にあろうとなかろうと，それを参照するための固有の識別子（補助記憶アドレス）を持つ．
2. ページの主記憶の記憶場所を変えたり主記憶から取除く時はただ1つの構造，すなわちハッシュ・テーブルだけを変更すればよい．
3. 特別の考慮なしで共用するページを適切に取扱うことができる．

この方式は実行中のプロセスがハードウェア・マップ中に含まれていなけれ

ば能率があがらない．なぜならば，ページ・テーブルを検索し，その後さらにハッシュ・テーブルを検索することは，ハッシュ操作過程を手助けする特別なハードウェアがなければ時間がかかるからである．もしページ・テーブルもハッシュ・テーブルとして構成すれば，2レベルのハッシュ・テーブル検索が必要になる．ページ・テーブルの場合と実記憶および論理記憶を関係付けるテーブルの場合にハッシュ操作を用いる動機は異なるということを認識することは重要である．前者の場合，すべての論理空間が使われていなければテーブルの大きさを小さくできる．後者の場合，情報を変更する時に多くのページ・テーブルによって変更する情報の1つのコピーを共用でき，更新を簡単にすることができる．

これで仮想記憶，補助記憶，および主記憶アドレスを関係付けるための方法に関する議論を終えることにする．ハッシュ・テーブルの利用は高速な検索ができるように変更する情報を持つテーブルを構成する手段として一般的な手法である．

書き込まれないページと書き込まれたページ　プロセスのスワッピングをさらに効率的にするため，プロセスのいくつかのページは**読出し**専用であり，プロセスの実行時にすべてのページが変更される記憶場所を含んでいるわけではないという事実を使うことができる．変更されていないページは**書き込まれていない**（clean）という．変更されたページは**書き込まれている**（dirty）という．読出し専用あるいは変更されなかった（書き込まれていない）ページはスワップ時に補助記憶に戻す必要はない．なぜならば，これらのページの正しい複写が既に補助記憶中に存在するからである．XDS-940ではこの概念はソフトウェアおよびハードウェアの記憶保護システムを使って実現されている．プロセスに対する実リラベリング・レジスタ類(マップ)を設定する時にすべてのページに対し読出し専用ビットをセットする．PMT内にはそのページが本当に読出し専用か否かを示すソフトウェア・コードが入っている．プロセスがページに書込みを行なおうとすると，記憶誤りトラップが発生する．システムはそのページが本当に読出し専用かどうかをチェックし，そのページが読出し専用でなければ，ページが書込まれたことを示すPMT内のビットをセットし，このページに対するリラベリング・レジスタ内の読出し専用ビットをクリアし，そして書込み命令の実行を許す．別のシステムではページが変更された時に各物理ブロックに関連したビットを自動的にセットするものがある．このビットはページが書き込まれているかどうかを判定するための命令を使ってテストで

きる.

まとめ　復習すると，物理記憶はそのページの各々について1つのエントリーを持った実記憶テーブル，あるいは実記憶カウント・テーブルといった機構により使用状況が記録されることを知った．RMT の各エントリーは実際にジョブのどのページがどの物理ブロックを使っているかを示すのに使われる．RMC はどの物理ブロックが共用されているかを示すのに使われる．XDS-940 プロセスの仮想記憶はプログラム記憶テーブルおよび共用記憶テーブルに関連して使われる擬似リラベリング・レジスタによって決められる．RMT, PMT, SMT，擬似リラベリング・レジスタ機構は仮想記憶と実記憶を関係付けるための技法の1つである．ページ・テーブルおよびハッシュ・テーブルを用いた別の方法についても議論した．この方法は概念的により簡単であるが，より多くの記憶領域が必要になる．

Multics にも本節で述べたものと同様なテーブルが物理記憶を記録するために設けられている．プロセスの仮想記憶はファイル・ディレクトリで始まる一連のテーブル内に記録される（6.1節で述べる）．すなわち，活用セグメント・テーブルはシステムに公開されたセグメントを記録するのに使われ，公開セグメント・テーブル，セグメント・ディスクリプタ・テーブル，およびページ・テーブルはプロセスに公開されたセグメントを記録するのに使われる．これらのテーブル類およびその使われ方は2.5節でやや詳細に述べた．

5.3.3 スワッピング

概　要　ここでスワッピング一般，特にデマンド・ページングにおいて遭遇する問題についてより詳細に眺めてみよう．仮りに主記憶が十分大きければ，スワッピングは必要ないであろうし，少くともずっと低い頻度で行えばよいはずである．主記憶の容量に関する現在の主要な制約はその価格にある．また信号伝幡時間も一要因になっている．しかしながら，ハードウェア技術は急速に変わりつつあり，主記憶の価格は下がり物理的な大きも小さくなると期待できる．

主記憶の価格が下がったとしても，磁気ドラムなどの補助記憶に比べれば1桁は高いはずである．そこで設計者はこの価格の差を利用して，スワッピングを活用するシステムを設計する．補助記憶機器には2つの重要なパラメータがある．すなわち，(1) 必要な情報ブロックをアクセスするのに要する平均時間，(2) 主記憶との間のブロック転送に要する時間である．前者は平均アクセス時

間(average access time)と呼ばれ，後者は転送速度(transfer rate)に逆比例する．スワッピングのために用いられる一般のドラムの平均アクセス時間は15〜40ミリ秒である．UNIVAC は容量 1.5×10^6 字，平均アクセス時間 4.25 ミリ秒の小型ドラムをもっている．転送速度はドラムによって異なるが，6ビット字で1秒当り数百Kから数M字である．上述の2つの時間の他に，ハードウェアの特性に関連するが，必要な情報の記憶場所を見つけるために余分な時間がかかかる．

スワッピングに関連する問題の一解決法は比較的平均アクセス時間が小さく，しかも転送速度の高い補助記憶機器を使うことである．本節では，大容量磁心記憶を用いた場合の解決法について述べる．ここで，スケジューラおよび記憶割付けシステムが解決しなければならない基本的な問題を見直してみることが有用であろう．

初期のシステムでは記憶割付けもスワッピングも非常に簡単な方法を採用していた．1つのプロセスの実行と他のプロセスのスワッピングを重複して行うことはしなかった．いつも主記憶内にはたった1つのプロセス全体しか入ってなかった．そのプロセスが切替えられる場合は，いままで実行されていたプロセスを完全に補助記憶に移し，次のプロセスを主記憶にロードしていた．全てのプロセスは，たとえば記憶場所0といった，ある定まった記憶場所に対して相対的にロードされた．すなわち，動的再配置は用いられていなかった．MIT のCTSS システムはこの方式を若干変更したものを用いていた．この場合，入ってくるプロセスが必要とする大きさの領域だけ前に実行していたプロセスを追出していた．記憶保護も用いていた．このような考え方は，最初のプロセスが再びプロセッサの制御を得る場合，仮想記憶の一部が依然として物理記憶内にあれば転送時間を節約できることにあった．この概念をCTSS技術ノートに基づいて，図5.4に示す．3番目のプロセスが2番目のものより大きければ，最初のプロセスは更にその分だけスワップ・アウトされる．

次の改善は動的再配置を導入することであった．これは主記憶内にいくつかのプロセスを同時に常駐できるようにしたものである．

動的再配置方式に続いて，1つのプロセスの実行と他のプロセスのスワッピングを重複して行う方法が採用された．更に**デマンド・ページング** (demand paging) の概念が導入された．これはプロセスが現在必要とするページだけを，しかも必要な時に主記憶の中にもってくる方式である．ページをもってくる間は，他のプロセスの実行を行なおうとするものである．この概念について本節で更に詳しく述べる．

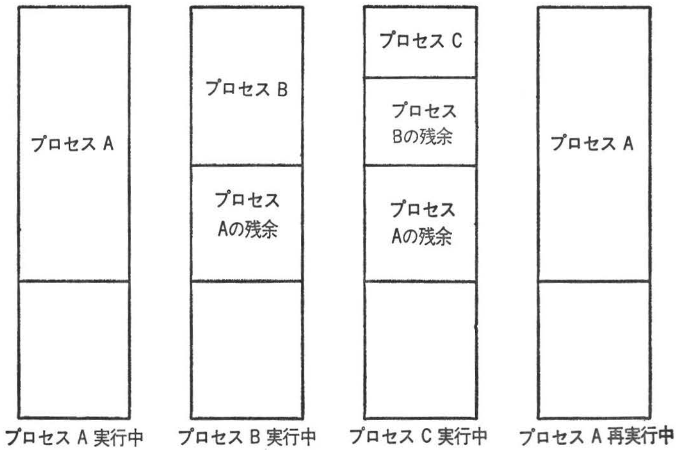


図 5.4 CTSS 記憶割付けアルゴリズムによるプロセス A, B, C 実行時の記憶装置の内容

実行とスワッピングの重複　実行とスワッピングを重複させた場合の一般的な問題点について考えてみる．以下の検討のために，次のような仮定をおく．すなわち，主記憶内にいくつかのプロセス，あるいはそれらの一部が入っているものとし，プロセスを実行するためには，その全てが主記憶になければならないものとする．実行とスワッピングを重複させたい場合，プロセスを実行させるために割りふる簡単な方法は次のとおりである．準備リストを参照し，最優先のプロセスを見つける．もしそれが主記憶内にあれば，実行させる．一部分しか主記憶にないか，あるいは補助記憶に全てある場合は，スワッピング指令を出す．スワッピングを行っている間は，主記憶内にある最優先プロセスの実行に制御を移す．

ここで直ちに読者は，スワッピングと実行を重複して行うためのスケジューリングおよび記憶割付け法を設計する人が直面する問題点のいくつかが分かったはずである．プロセスがスワップ・インされる場合，入ってくるプロセスのために主記憶内に十分その余地をつくるが，そのときどのプロセス，またはプロセスのどの一部をスワップ・アウトすべきであろうか？　1つの解法は，スワッピング中に実行すべき主記憶内のプロセスのページに印をつけ，ある簡単な方法に従って追出すページを選択することである．最も一般的に用いられて

いる方法は最も最近実行したプロセスのページを追出することである[27]．しかしながら，実行すべきプロセスが実行直後にブロックされ，そのときスワッピング進行中であったらどうなるであろうか．主記憶内にある最優先の次の順位をもつプロセスはそのスワッピング中にいくつかのページを失う可能性がでてくる．入ってくるプロセスのための余地を作るのに，スワップ・アウトすべきプロセス，またはプロセスのページを選び，しかも上述の問題を解決するような方法を考えることでできるが，ずっと複雑なものになってしまうであろう．記憶割付けの計算は極めて大変な仕事であり，システム上重大な浪費をもたらす．

ある有用な計算をスワッピングと重複できるように保証しようとする方法は，実行が始まったらすぐにはブロックしそうもない計算主体のプロセスをいくつか常時主記憶の中に入れておくことである[21,23]．一般に，5.2.6 節で論じたような多重レベルのスケジューリング方式を採用していれば，このような計算主体のプロセスはシステムが識別できる．更に計算を続けるであろうという仮定の下に，長時間計算し続けているプロセスを1つ主記憶の中に入れておく．

プロセスがブロックし，準備リストからスワップ・インされるプロセスが選ばれると，入ってくるプロセスを未使用のページ，または主記憶中にある最底優先順位のプロセスで，計算主体でないもののページに割付ける．いくつかのプロセスが共用しているページは最後の手段としてスワップ・アウトすれば，実に能率がよくなるはずである．なぜならば，共用しているページをすぐに追出してしまえば，いくつかのプロセスを主記憶中に完全に存在する状態からただちに取り除くことになるからである．入ってくるプロセスのための余地を作るためにスワップすべきページが決まると，スワップが行なわれている間，主記憶中に存在する最優先のプロセスを実行させる．実行中のプロセスがブロックすれば，主記憶中にある次の優先順位のプロセスに制御を移す．このプロセスがブロックすれば，その次の優先順位をもつ主記憶中にあるプロセスに制御が移され，計算主体のプロセスに制御が移るまで，同じようなことが行われる．スワップが終れば，制御は記憶装置に持ち込まれた新しいプロセスに移される．

計算主体のプロセスはブロックするまで主記憶に留まることが許される．ブロックした時点で，他の計算主体のプロセスに替えられ，準備リストの適当な場所に入れられる．上述のような概念的な方法はスワッピング中でも効率的なプロセッサ利用を保証しようとするものであり，スケジューリングと記憶割付けは密接な相互関係があることを示している．このような考え方は，ページン

グはしていなくても，ベース・レジスタのような動的再配置機構をもつシステムには適用可能である．

デマンド・ページングの問題　ここで，デマンド・ページングに附随する複雑さについて調べよう．スワッピング・アルゴリズムの選択およびデマンド・ページ・システムの成功，不成功は次の事に関係する：すなわち，

1. プログラムがページを要求する特性
2. ソフトウェアで補助記憶中のページを見つけ出す時間
3. バックアップ記憶として使われる補助記憶のアクセスおよび転送特性

プログラムのページ・アクセス特性に関しては数多くの研究がなされているが[5-8]，それによると，プロセスが実行を開始した後はほぼ図 5.5 に示すようなプロセスのページ．アクセス特性があることを示している．タイム・スライスのごく初期ではプロセスはほとんど自分のページをアクセスしている．これは，デマンド・ページ方式では実行が開始された直後の短かい時間に高いページの往来があることを意味している．Fine の研究では[5]プログラムの大きさが14Kから44K語にまたがり，平均30K語であるやや大きなプログラムを5つ調べている．各プログラムは1K語のページに分割されるものと考え，どのプログラムもページ化した計算機を意識して作られていないとしている．Fine はタイム・スライスのごく初期（30ミリ秒）では平均して18から20ページをアクセスすることを見出している．彼は実際に，プログラムが20ページを必要と

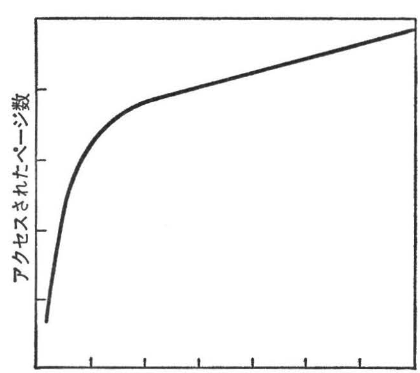


図 5.5 ページ・アクセス特性

すれば，これらのページをプログラムが要求する時間は25％の7ミリ秒以内であることを明らかにしている．このような高いページの要求はデマンド・ページングのスワッピング・アルゴリズムを採用すれば，プロセスが1ページを待つのにしばしば停止させられることを意味している．このため，デマンド・ページ方式を採用して効率のよい動作を達成するのは悲観的になってしまう．現在までの研究で収集されたデータはページ化された計算機を意識して書かれたプログラムから得たものでないが，仮りにページ化した計算機を意識したプログラムであっても同じような結果であろうと Coffman は云っている[6]．一方，Denning はこの意見に納得していない[9]．結局，データで他の結果を示さない限り，プログラムは 図 5.5 に示すような特性をもつものと設計者は仮定してもよいと思われる．

プロセスのページが主記憶にもち込まれ，実行が開始されると，そのプロセスは主記憶にないページをアクセスするまで短時間活動するが，その後は主記憶に必要なページが取り出される間活動が中止される．主記憶の持ち込まれたページが多ければ，活動期間は長くなる．活動していない期間に他のプロセスの実行を重複させるには，他のプロセスのいくつかはそのほとんどのページを主記憶にもっていない限り，あるいは要求するページに対するアクセス時間に転送速度が極めて短かくない限り，うまく行かないであろう．デマンド・ページングを採用する場合，必要とする主記憶の大きさと補助記憶へのアクセスの速さとの間に重大な関係があるように思われる．これを考えないと，システムはすぐにそのほとんどの時間をスワッピングに費すことになり，ユーザ・プロセスを実行する時間が少くなってしまう．Fine は最初の10活動時間に全体の平均活動時間は0.8ミリ秒（SDC-Q 32 計算機）であることを見出している．仮に平均アクセス時間17ミリ秒，転送速度無限大のドラムを使ったとすれば，0.8ミリ秒の実行を達成するために必要となるページを待つ，いわゆる活動していない時間は170ミリ秒にもなる．システム中の他のプロセスも全て同じような統計値で振舞い，I/O サービスで停止してしまうであろうから，たとえ高速ドラムからのデマンド・ページングであっても，計算をしない，情報転送を行うシステムとなってしまう可能性がある．初期の IBM 360/67 システムに関する Nielson のシミュレーション[10,11]も同様の結果を示している．デマンド・ページングにこのような問題があることを頭に入れ，次にリソース割付け問題をより一般的に調べ，デマンド・ページングの問題に対して既に実現あるいは提案されている解決法を検討しよう．

Randell と Kuehner[13] が指摘しているように，リソース割付け法には2つ

の似かよった方法がある．(1) プロセスの実行期間中に必要とされるであろう全てのリソースをそのプロセスに前似つて割付ける，(2) 必要になった時点でプロセスにリソースを割付ける．プロセスの実行を始める前に，プロセスの全アドレス空間，またはアドレス空間の大部分をスワップ・インすることは，記憶割付けに関して上述(1)の方法を採用したものである．デマンド・ページングは(2)の方法である．ページングを用いればプロセスを適当な単位で取扱うことが許され，2つの方法の釣合いをとった方法を開発することができる．上述の似かよった方法は共に良いとも悪いともいえない．大ざっぱな解析では，方法(1)は(2)ほど良くない傾向がある．リソース割付け法の選択は次のことを考慮に入れて行う必要がある[13]．すなわち，

1. 非生産的なリソース利用の価格
2. リソース要求が満されるまで待合わせる際の損失
3. リソースの要求が起こる頻度
4. 他のリソース利用時に別のリソースを割付ける方法の効果
5. リソース割付け法それ自身を記憶し実行させるためる記憶空間と時間の損失

これら全ての要因がいかに相関があるかを知るには，特定のシステムの下で十分検討を加えてみる必要がある．これだけのことを知っていれば，デマンド・ページングの検討を更に進めることできる．上でしばしば論じた方法でデマンド・ページングが効率よくなるような種々の着想が生れてこよう．

デマンド・ページングの問題に対する解決法　デマンド・ページングの問題に対する解決法にはいくつかの提案があるが，比較的重要なものは次のものである[9]：

1. ユーザまたはコンパイラの最適化の段階でプログラムの構造を改良し，ページ要求の比率が小さくなるようにしたり，システムがプロセスの構造に関してユーザの助言を受けられるようにする．
2. プロセスのスケジューリングおよび主記憶割付け法を改良する．
3. スワッピング媒体への平均アクセス時間を短かくする．

上述の提案1を実際に行って成功した例がある．その一例はページ化したリスト処理言語 LISP に関する Bobrow の研究[30]であるが，更に検討を加える必要がある．

Denning[9] は次の理由によって，ユーザからは有益な助言は期待できないことを指摘している．すなわち，

1. ユーザは他人の仕事の上に自分の仕事を構成することがあり，共用するプロセスがいかに構成されているかを知らなかったり，アクセスの構造がデータに依存していることがある．
2. ユーザからどのような助言を受けるべきかが明白でない
3. ユーザが与えた助言はそのユーザのジョブのためにシステムを最適化しようとするが，システムは全てのジョブを考慮しなければならない．

Denning は実際に，2つの理由からコンパイラが構造に関する情報を提供することは期待できないと指摘している．その理由は，(1) プログラムはモジュール構造が多く，種々のモジュールの構造はコンパイル時に利用できない．実行時に初めて決まるようなデータに依存した構造がありうる．(2) 構造を注意深く追跡する仕組をもったコンパイラは非常に遅く，しかも大きくなる．したがって，プログラムの開発中の高速コンパイルを要求するユーザには不満足なものになることである．このことから，Denning は上記提案2である，いままでに観測したプログラムの振舞い特性に基づいて割付け法を決めるか，あるいは提案3のより高速アクセス機器を使わなければ，デマンド・ページングの概念は現実的なものにならないと主張している．

オペレーティング・システムの構造をページ要求の比率が少なくなるように改善することはずっと有望である．第7章で述べるように，オペレーティング・システムのページ・アクセス特性に関して統計をとることができ，オペレーティング・システムのモジュールをページ要求の比率が最小になるように再ロードすることができる．

記憶割付け法の概要　記憶割付けおよびスケジューリングを行うときには主に次の2つの時点がある．(1) ページが要求されたとき，および，(2) 入ってくるページのための余地を作る上でページを追出すとき，である．ここで暫くの間，ページが要求された時に初めて1ページをもってくるという方法で (1) を扱うことを仮定しよう．以下でこの点を振り返ってみる

ページを追出す過程をしばしば，**ページの入れ替え** (page turning) あるいは**リプレースメント**(replacement)と呼んでいる．追出すべきページを選ぶアルゴリズムがリプレースメント法である．このような方法を選ぶ際の問題点のいくつかは5.3.3節で触れた．リプレースメント法の目標は一般に，近い将来の

参照が最も少ないと思われるページを追出すことにある．いま，主記憶はいくつかのプロセスのページでいっぱいになっていると仮定しよう．最も単純な方法が2つある．その1つは**先入先出**（first-in first-out）（FIFO と簡略）と呼ばれる方法である．新らしいページが要求されると，いつでも最も古い時期にもち込まれたページを追出す．もう1つの方法は LRU（least recently used）と呼ばれる方法で，新しいページが要求されると，いつでも最も長い時間参照されなかったページを追出す．もう1つ，もっとこみ入った方法があるが，これには重要な概念が多く出てくる．それは**ワーキング・セット**（working-set）（WS と簡略）**法**と呼ばれ，LRU 法に関係があるが，現在実行しているプロセスからのみページを入替える．

FIFO と LRU 法　FIFO 法は実現が容易なのが特徴である．ページが主記憶にもち込まれた時にページ識別子を FIFO リストに置くようにすれば実現できる．LRU 法は実現がやや難かしいが，過去において最も長い時間使われなかったページは将来においても最も使われないページであるという考え方がよいように思われる．

LRU 法と実現する方法——いずれも非常に似かよっている——は既に数多く実用に供されたり，提案されている．基本的な考え方は次のとうりである．主記憶の各ブロックにカウンタを1つ付随させる．ページが参照されると，前似って定められた値を対応するカウンタにセットする．周期的に全てのページのカウンタから1を引く．最も長い間使われなかったページは最も小さな値が入っているカウンタをもっている．これらカウンタはしばしば**経時レジスタ**（aging registers）と呼ばれるが，カウンタの内容を1減らすということはページが年をとったことを意味している．

実用システムでは，カウンタは有限の長さのものである．したがって，0の値のカウンタをもつページは全て同じ年と考え，一番最初のものをリプレースメントの候補とする．この考え方を実現する一方法は物理記憶に別のハードウェアを付加することである．たとえば，ある記憶ブロックが参照されると，カウンタがセットされたり，コンデンサが充電される．時間が経つとカウンタは一定時間毎に内容が減らされ，コンデンサは電荷を失って行く．ディジタルにしろアナログにしろタイマが0になれば，そのページが印され，追出しページの候補となる[9]．

物理記憶の各ブロックに，経時レジスタと呼ばれるいくつかのビットを対応づけ，ソフトウェアで同じ機構を実現することができる．概念的には次のような

方法をとる．ページが参照されると，経時レジスタの左端のビットに1がセットされる．また，定期的に全ての経時レジスタを右に1ビットだけシフトする．シフトに際しては左端から0が入り，右端からはみ出した1は失なわれる．この方法によれば，一定時間内に参照されたことのあるページは全てその経時レジスタ内に少くとも1ビットは1がセットされている．図5.6はページ参照前，ページ参照後，および経時（シフト）後の経時レジスタを示したものである．

経時レジスタの考え方をソフトウェアで実現する場合に多くの問題が起こる．(1) シフトとシフトの間で各ページに対して起こる最初の参照をシステムが検出しなければならない．このような表示をするには，記憶保護システムを記憶テーブルと関連して利用することができるが，システムには余分な負荷がかかる．書込まれないページと書込まれたページに関する討論（5.3.2 節）はこのような機構の基礎である．GE-645 はページ参照を検出するハードウェアを備えている．主記憶の各物理ブロックに付随して参照ビットが1つついている．ページが参照されると，対応する物理ブロックの参照ビットがセットされる．このビットは命令語でテストすることができ，主記憶内のページの経時測定のために用いるサンプリング時に，そのハードウェア参照ビットが経時レジスタに移され，参照ビットはリセットされる．(2) 全ての経時レジスタをシフトすることは，経時間隔が非常に短かい場合はかなり大きな負荷になる．(3) 最も長時間使われなかったページを見つけ出すために経時レジスタを探ることは時間の浪費となる．シフト間隔と経時レジスタの長さを慎重に選べば，これらの問題は最小限度にとどめることができる．

Corbato は経時レジスタがシステムの効率に及ぼす効果を研究するために，上述の考え方を変更したものを使っている[16]．すなわち，

1. 追出し候補のページを循環リストにのせる（循環リストとは最も新しいエントリーが最も古いエントリーを指しているリストである）.
2. 主記憶の各ブロックは経時レジスタを付随してもっているが，その左端のビットはそのブロック内にあるページが参照されたとき1にセットされる.
3. 一度にリスト上の1ページだけがその経時レジスタを更新（右1ビッ

| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |　| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |　| 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

図 5.6　経時レジスタ．(a) ページ参照前；(b) ページ参照後；(c) シフト後

ト・シフト）させ，その経時レジスタが0になったかが調べられる．

4. 主記憶にない新しいページが要求されたときのみ経時動作が起こる．

5. 0の値をもつ経時レジスタのうち，最初に見つけ出されたものを入替える．長さ0の経時レジスタは FIFO 法となる．

Corbato は長さ0のカウンタと長さ1のカウンタの中間でシステムの効率がかなり改善されることを見出した．カウンタの長さを1以上に増しても重要な性能改善をもたらしていない．Corbato が実際に用いた記憶割付けアルゴリズムは上述のものを更に改良したものであるが，ここでは経時動作の概念，経時カウンタの長さを選ぶ設計上の問題，および経時区間を決定する方法の説明を述べているにすぎない．

FIFO および LRU 法に伴う問題は主記憶中に存在する全てのプロセスにとって共通である．すなわち，主記憶中に存在する全てのプロセスのページは追出し候補と考えられる．このような方法はプロセス間で好ましくない交信を引起こす．それはあるプロセスがもつページを追出し，しかもスワッピングと重複して処理を実行させたとき，他のプロセスのページが必要となった場合である．この場合，どのプロセスも別のページを要求せずに長い間プロセスを実行できるほど十分なページを主記憶中に蓄えていないという事態が起きうる．必要とするページが持込まれている間に，そのプロセスの他のページが失なわれることもある．それは経時動作（LRU）または FIFO ページ・リストの位置に起因して，他のプロセスを暫くの間実行するために必要なページの余地を作るためである．このような状態を**スラッシング**（thrashing）[9,14] とたくみに呼んでいる．

WS法　あるページが他のプロセスのページのために主記憶から追出されるとき，少くともいくつかのプロセスは手をつけないでそのままにしておく方法で追出すページを決めるべきであると前に述べた．手つかずのままにしたプロセスはスワッピングと重複して実行できる可能性がある．Denning はデマンド・ページングのためにこの考え方の1つを提唱し，**ワーキング・セット**（working-set）法[9]（WS 法と簡略）と呼んだ．WS は以下で分かるように，実現に際していくつかの難かしさがあるが，実用的なデマンド・ページング方式が得ようと努めるべき目標のように思える．WS 法の考え方の基は図5.5の観察にある．主記憶中にいくつかのページが出された後は，ページ要求の比率は急激に減少することが分かる．プログラムはいくつかのページの組である期

間動き，その組は一度できると適当な割合で変わるから，このようなことになると思われる．

Denning はある時点で要求されるページの組を**ワーキング・セット**（working set）と呼んだ．彼はワーキング・セットを決める方法を提案したが，LRU 法において用いた方法と同じである．また，彼はこの情報をプロセッサと記憶の割付けにいかに使うかを示している．実際には，WS法は現在実行しているプロセスのページだけを追出し候補と考えたLRU 法の一形式と考えることができる．時刻 t におけるプロセスのワーキング・セット情報 $W(t, \tau)$ は処理期間 $(t-\tau, t)$ にプロセスが参照したデータ項目の集りを定義している．パラメータ τ は主記憶の大きさ，補助記憶の転送速度，およびアクセス時間などのパラメータが反映されるように選ばれなければならない．ワーキング・セットにおけるページ数が $\omega(t, \tau)$ である．最初の問題は $W(t, \tau)$ と $\omega(t, \tau)$ を決めることである．1つのプロセスのワーキング・セットは前節で述べた経時動作技術用いて決定される．しかしながら，現在実行中のプロセスのページだけしか考慮の対象としていない．

いま，何らかの方法でワーキング・セットとその大きさが分かったとしよう．その時この情報をスケジューリングと割付けにどのように使えるだろうか？　Denning の提案は，(1) 現在のワーキング・セット内のページ数と同じ領域が主記憶に空いていなければ，いかなるプロセスも実行のために主記憶にロードしない，(2) リプレースメント法として，実行中のプロセスのページで，現在のワーキング・セットにないものを先に追出すことである．この方法の目標はワーキング・セットのページを含んだプロセスだけを主記憶にとどめ，リプレースメントに際しては互のワーキング・セット内のページをプロセスが追出すことを極力避けることにある．この方法は実行とスワッピングが重複できる可能性を高めている．入ってくるプロセスのワーキング・セットのための余地を作るためには，現在主記憶にある他のプロセスのワーキング・セットを追出す必要がある．プロセスが主記憶に沢山あって，そのワーキング・セットが入る余地がないときは**空間強取**（space-squeezed）状態が起る．この場合，主記憶には前節で述べたスラッシングが起る可能性がある．Denning の提案は直観的に見て，ワーキング・セットを記録する犠牲を除けば，問題に対する解法であると思われるが，ワーキング・セットの記録はシステムにとって重荷となり，その着想は今までに実用となっていない．しかしながら，主記憶中にあるプロセスの記憶要求の相互作用を最小限にとどめようとする考え方は重要で，より容易に実現する方法を目指すことが目標というべきであろう．

Denning はワーキング・セットの余地がなければ，いかなるプロセスも実行を始めないことを提案したが，図 5.5 を照してみれば，実際にはページに関するその前のワーキング・セットが主記憶に戻されなければプロセスの実行が始まらないことが分かろう．この概念を**同族ページング**（affinity paging）または**ページのプリローディング**（preloading）と呼んでいる．Denning はプロセスがブロックまたは交信を終了したとき，次に必要とするワーキング・セットはその前のものとかなり異なっており，したがって不必要なリソースを浪費して，余分なページが持込まれることがあると述べている．前以ってワーキング・セットをもってくるという議論は，プロセスがブロックまたはユーザ要求のプロセス・サービスが終了する前に，1クォンタム時間以上プロセスが実行されることと考えられる．それゆえ，実行の最初のワーキング・セットは多くの場合，その前に実行を終了したワーキング・セットと同じものである．これは実行の前にその前のワーキング・セットを戻し，この時点からデマンド・ページングを用いるべきであることを示唆している．5.2.3 節で述べたように，ほとんどのプロセスは1クォンタム時間の中に終了するように，クォンタムの大きさを調整すれば，この議論は不要になってしまう．

要約すれば，Denning はワーキング・セットに基づいたスワッピング・アルゴリズムを用いているシステムにとっては，プロセスのスケジューリングと記憶の管理は相互に関連するという重要な指摘をしている．ここでの重要な示唆は主記憶と補助記憶のパラメータは基本的なスワッピングの考え方の性格はどのようなものであるかを示していることである．補助記憶のパラメータの選択は他の重要なリソース割付けアルゴリズムにも大きな影響を与えることになる．

大容量磁心記憶の利用　ここでスワッピングの性能を改善する上述の3番目の方法，すなわち高速アクセス補助記憶の利用について考えてみよう．比較的高速なスワッピング媒体として，**大容量磁心記憶**（bulk core）のようなものが用いられている．大容量磁心記憶は等速記憶であるが，主記憶よりアクセス時間が大きい代りに価格が安い．回転機器の平均アクセス時間を UNIVAC のドラムの4ミリ秒以下にするには恐らくドラムをもっと小型化し，情報記録密度を更に高める努力が必要となろう．さもなければ，このようなドラムを多数必要とすることになり，大容量磁心記憶に比べてドラムの価格が安いことの利点が失なわれてくる．ある程度改善を期待できるが，2倍がよいところだろう．大容量磁心記憶は回転機器に比べてやはり高価であるが，システムの性能

全体から見れば経済的になる．ディスク記憶装置は読み書きヘッドの位置合わせのために平均アクセス時間が比較的長く，しかも一般に転送速度が低いが，これをスワッピング媒体に選ぶのはうまい選択ではない．もっともドラムの特性をもつトラック毎に固定ヘッドを備えたディスクは使える．

ここでは，大容量磁心記憶の利用について簡単に考えてみるが，大容量磁心記憶は特に価格が下がれば，その利用はより一般的になると期待できる．現在では，基本的に2種類の大容量磁心記憶がある．1つは IBM の**大容量記憶**(LCS : large-capacity storage) であり，他は CDC の **拡張磁心記憶**(ECS : extended-core storage) で代表される．これら2つの記憶装置はバック・アップ記憶として異なった性格をもっており，それぞれ異なったスワッピング・システムを実現することになる．Carnegie-Mellon 大学では，IBM 360/67 のスワッピング媒体として LCS を利用する実験をしている[17,27]．LCS はサイクル時間8マイクロ秒，アクセス時間4マイクロ秒で CPU から直接アクセス可能である．また転送は記憶チャンネルを介してブロック単位で行われ，LCS を平均アクセス時間8マイクロ秒，転送速度毎秒 400K 32 ビット語のドラムに見えるようになっている．LCS システムでは，実際にページの大きさは2種ある．1つは 1,024語で，チャネルを経由してスワップされる．他は LCS の倍長語で直接アクセスできる．モニタが1ページ内のほんの数語だけしかアクセスしないことを知っていた場合は，スワッピングを行うよりも，LCS 内のその語を直接アクセスした方が安上がりである．モニタがそのような事実をいかにして知るかは未解決の問題であるが，そのような機会は存在する．Fikes ら[27] はこの問題を検討しているが，彼らはシステムから見た価格（システム価格）には LCS から主記憶に1ページを移す価格と LCS 内で直接実行する価格とがあることを指摘している．タイム・スライス中に1つのページを何回かアクセスすることがあるが，そのページを LCS から主記憶に移すか，あるいは LCS 内で直接実行するシステム価格が等しくなるようなアクセス回数が存在する．価格を厳密に定義すれば，このアクセス回数は計算できるはずである．

このアクセス回数が与えられ，Fikes らが各ページの**アクセス密度** (access density)と呼んだものを測定できれば，命令を実行するためにそのページを主記憶に移すべきか否かについて適切な判断をシステムは行なえる．あるページのアクセス密度とはタイム・スライス当りそのページにアクセスする平均回数である．この平均値を計算する場合，タイム・スライスの回数は，そのページが少くとも1回アクセスされるようにとる．判断に加える他の要因としては，主記憶内にそのページのための余地があるか否か，すなわちあるプロセスがそ

のワーキング・セットからあるページを失うことになるか否かである.

Fikesらはアクセス密度を決定する方法を与えていないが，LCS内で実行するか，あるいはスワップすべきかを決める直観的な基準は検討している．第一近似としては，システムのページおよび共用ページは全てスワップすべきだと云っている．ユーザの専用プログラムおよびデータのページはLCS内で直接実行することも考えられる．この問題に関する明確な判断規則を設定するにはもっと研究を重ねる必要がある.

1K語のページをLCSからスワップするには2.5ミリ秒必要であるが，プロセスがページ要求を出すのに時間がかかるとすればLCSの平均ち待合わせ時間は約3～4ミリ秒となる．この値はドラムよりもよい．また，価格が5分の1以上低下すれば，LCSはスワッピング媒体として経済的にドラムと匹敵するであろう．Carnegie-Mellon大学におけるLCS利用の経験でも期待通り，重要な性能改善が示されている.

大容量磁心記憶を使ってもスワッピングの問題を完全に解決しているわけではない．プログラム・サイズに制約を設けないで，全てのユーザのプログラムおよびデータを大容量磁心記憶に蓄えることは経済的に不可能であり，大容量磁心記憶とドラムまたはディスクとの記憶階層において単一レベルにするスワッピング問題が再び出てくる．Fikesらは大容量磁心記憶を割付ける問題を若干詳細に論じているが，概念的には主記憶を割付ける際に出てきた問題に似かよっている．LCS内のページを入替えなければならないとき，最も使いそうもないページを追出したい．Fikesらは準備リストにプロセスを優先順位にならべ，更にプロセスがブロックされている時間を基準にならべかえている．すなわち，最優先プロセスは準備リスト上の最優先プロセスであり，最低順位プロセスは最も長い間ブロックされていた最低順位のプロセスである．最低順位プロセスのページが先に追出されることになるが，参照されなかったプロセスの記憶空間にあるページ，または読出し専用，すなわち書込んではいけないページを他に先がけて追出すように改善されている.

LCS利用に関するもう1つの考え方は，全くスワッピングをせずに全てのユーザ・プロセスおよび使用頻度の低いシステム・プロセスを大容量磁心記憶に残し，大容量磁心記憶から直接実行するようなシステムに見られる[18,19]. Pittsburgh大学のIBM 360/50システムはこの方式を採用し，会話形言語インタプリタを高速記憶に入れ，ユーザ・プログラムを大容量磁心記憶に蓄えているが，その観測によるとインタプリタとユーザ・プログラムを共に高速記憶に入れた場合に比較して，プログラム実行速度は20%減であることが分かって

いる．CPU がもっと速ければ，速度の低下はより著しいはずである．ハードウェアの処理装置の速度が大容量磁心記憶とほどよく調和していれば，大容量磁心記憶から直接的に実行するために起る効率の悪さはスワッピングによる効率の悪さ以下，あるいは同程度であろう．また，大容量磁心記憶から直接実行すればシステム設計が簡単になると思われる．

CDC の拡張記憶は CPU から直接アドレスできないが，最初の60ビット語をアクセスするのに3ミリ秒かかる．いったん最初の語が見つかると，転送速度は毎秒10M 60 ビット語が維持される．したがって，1K語ブロックは 100 マイクロ秒で転送できる．このような高速性は2つの意味がある．すなわち，(1) 転送に全記憶サイクルを必要とするから，スワッピングと計算を重複させることはできない．(2) ECS 内にある必要なブロックを見つけ出すための処理はそのブロックをアクセスする際の遅れの大部分を作る．

ECS システムは転送速度が高速なので，情報の転送単位としては，ページよりはプロセスまたはセグメントの方が向いている．したがって，ECS を備えた計算機はドラムや LCS を設けた計算機よりも記憶管理システムがずっと簡単にできるはずである[35].

まとめ　記憶割付けの問題は経済的および物理的制約からシステムが利用できる主記憶の大きさを制限することになる．記憶割付けは技術的な問題である．プロセスをスケジュールするアルゴリズムは記憶割付け法にも密接な相互関係がある．記憶割付けにおける基本的な問題には2つある．すなわち (1) 物理記憶の利用，および物理記憶のどこにプロセスの論理空間が格納されているかの記録をとること，(2) 利用できる主記憶を相争うプロセスに割付ける判断規則を作ること，である．後者をスワッピングの問題という．割付け法には，(1) プロセスの論理空間の全てに対する物理記憶を割付けるか，あるいはその手ごろな一部を割付けておいて，実行の前に主記憶に論理空間をもってくる方法から (2) 1ページの大きさに相当する量の記憶を割付け，必要に応じて割付けを増やしていく方法に至るまで種々のものがある．後者の概念をデマンド・ページングと呼んでいる．

いずれの割付け法もその目標はあるプロセスの実行を他のプロセスのスワッピングと重複させることにある．この重複を可能にするためには，主記憶に入ってくるプロセス，またはその一部の空間を作るために，追出すべきプロセス，またはその一部を決めるリプレースメント・アルゴリズムを上手に選び，いくつかのプロセスの記憶要求が相互に重大な影響を与えないようにすべきで

ある．

スワッピングを改善する方法として，オペレーテング・システムを調べた他に，高速補助記憶の利用について論じた．また，プログラムの構造を改善する技術についても述べた．高速補助記憶，特に大容量磁心記憶の利用はいろいろ調べた問題の多くを解決し，システム設計を簡単にするように思われる．現在の経済的，技術的な傾向からすれば，本節で考えた問題の多くを軽減する種々の開発が行われるものと期待できる．

5.4 システム保護

5.4.1 概　　要

われわれの観点では，保護システムの設計はタイムシェアリング・システムの最も重要な点であるように思われる．システム保護はハードウェアおよびソフトウェアの双方に多くのレベルで存在する[2,20,34]．保護の問題は本質的および技術的の2種類がある．本質的問題は主にファイル・システムに含まれ，情報の秘密を守り，それを共用することにある．この領域についてはさらに第6章で論じる．装置の利用率の均衡をとって必要な経済性を達成するために，多数の仮想プロセッサが限られた物理的構成を共用することから，技術的問題が生じる．本節ではまず技術的問題について考えてみよう．基本的な問題はユーザ・プロセスからシステム・プロセスを保護し，またユーザ・プロセス相互の保護を行なうことである．読者はこの時点で4.1節を復習してみるのがよかろう．そこでは保護を主にハードウェアによる保護という観点で紹介した．

保護を2通りの方法でみることができる．(1) リソース自身(ソフトウェアならびにハードウェア) の保護，および (2) リソースへのアクセス経路の保護である．うまく設計されたシステムはおそらくこの2つの観点に集約された方法がとられているだろうし，システム全体を通して一様でかつ明確に実現されているだろう．またうまく設計されたシステムは最小レベルの制御でその設備への多数のレベルと多くの形式のアクセス権を提供している．

5.4.2 保護レベルの必要性

Multics の保護方式は概念的に重要である．なぜならば，システムを通して保護が非常に一般的で一様な方法で扱われているからである．一方，940 システムでは保護は多くの点で行なわれるが，集約した形では行なわれていない．

Multics リング方式（後で述べる）は多くのレベルの保護を提供するのに対し，940 ではたった2レベルしか行なわない．XDS-940 システムでの経験からみると，多数のレベルの保護を持つ能力があることが望まれる．XDS-940 システムにおける2つの基本的な保護レベルは，監視状態（executive status）と非監視状態（nonexecutive status）である．監視状態にあるユーザはすべてのシステム・ルーチンにアクセスでき，種々のテーブルおよびその他のパラメータを変更できる大きな力を持つ．この理由から，ユーザにはこの状態は与えられないが，それでも何人かのユーザは現在利用できないあるシステム設備を有効に使える．しかしながら，監視状態のユーザはシステムに勝手にアクセスできるので，このことに含まれる危険性は非常に大きい．この危険性があるために，一般にどんなユーザにも監視状態を与えるわけにはいかない．システム・プロセスは種々の保護レベルで有効に実行することができる．このような方法はハードウェアおよびソフトウェア誤まりで起こされる損害の範囲を制限することによって，システムの信頼性を上げることになろう．また，ユーザ・プロセスも多層の保護を利用してその信頼性を上げることができる．

5.4.3 テーブルによる保護

ソフトウェア・システムの中では，特定のリソースに対する保護情報はそのリソースに関連する種々のテーブル内に保持される．たとえば，第6章で述べるように，ファイルは**ファイル・ディレクトリ**と呼ばれるテーブルの中に格納された情報で保護される．Multics の場合，セグメント（Multics ではファイルとセグメントは同一である）に対する保護情報はそのセグメントが使用される時にそのセグメント・ディスクリプタ・テーブルに移される．XDS-940 と同じように，サブシステムへのアクセスもユーザの課金テーブルおよびそのサブシステムを初期化するために使われるテーブルの中に格納された情報によって制限される．XDS-940 のユーザはアクセスするサブシステムを定義するサブシステム状態を持つ．6.2 節で述べるように，I/O 機器は機器制御ブロックおよびファイル制御ブロックに格納された情報によって保護される．あるユーザの課金テーブルはそのユーザが磁気テープ装置あるいはグラフィック・ディスプレイへアクセスするかどうかを示しており，あるいは同様な情報がその機器に関連したテーブルに保持される．最も一般的な問題は，プロセスのアクセス権をお互いに関連して決めることである．

5.4.4 Multics における保護

プロセス相互のアクセス権を決めるのに，一般的で，集約され，しかも明確に定義された方法を Multics にみることができる[20]. 第4章で述べたように，Multics のハードウェアはセグメントへのアクセス経路ではなくセグメント自体を保護する．それゆえ，そのソフトウェア・システムはアクセス経路を制御する機構を持つ必要がある．あるセグメントから他のセグメントへのアクセス経路の各々を制御することはせず，セグメントはいくつかのクラスにまとめられる．制御されるものはクラス間のアクセス経路である．その保護方式は保護の同心円状のリングに基礎を置き，内側にあるリング程高い優先順位を持っている．セグメントはこれらのリングに割当てられる．最も重要なシステム・セグメント（ハードウェアとのインタフェース）はリング0にあり，次に重要なシステム・セグメントはリング1で，順次そのように割当てられる．ユーザ・セグメントはこれらより外側のリングに割当てられる．システムは大体次のように働く（その概念を図 5.7 に示す）．セグメントは同一リング内にあるその他のどのようなセグメントに対しても直接的にアクセスできる．もしリングの境界を横切ると，システムへの呼出しが作られる．あるセグメントからそれより高いリング番号にあるセグメントへのアクセスは自由に許されるが，低いリング番号のセグメントへのアクセスは制限される．内側のリングの呼出し

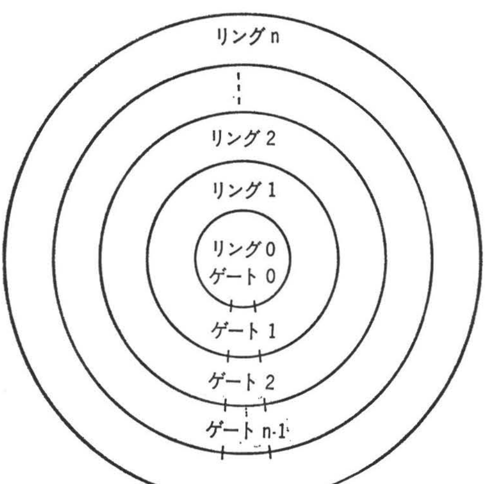


図 5.7 Multics のリング構造保護概念

は，**ゲート**（gate）と呼ばれるある入口点からのみ許される．システムは目的のアドレスを目的のセグメントに対する正当性を示すエントリー・リスト上のエントリーに対してチェックする．もしそのプロセスがアクセスを許されており，目的入口点が正当であれば，次のチェックが行なわれる．アドレスは引数として渡されるので，呼出しているセグメントが参照しようとするセグメントをアクセスできるかを調べるために，そのアドレスをチェックする必要がある．内側のリングは大きなアクセス権を持ち，これらのアドレスは時にチェックされないので，このことが必要になる．もし不適当であれば，引数として渡されたアドレスは内側のセグメントに誤まった動作を起こさせ，そのセグメント自体あるいは他のシステム・セグメントに相当な損害を生じさせることがある．

リング・システムにはその他にも複雑なところがある．それは2つのプロセスがセグメントを共用でき，正当性を確認した後一方のプロセスがブロックされ，確認されたアドレスを変更できる他方のプロセスを開始できるからである[20]．Graham はたとえ長い間割込みを禁止できる場合でも，内側のリングにある呼ばれたプログラムが確認されたアドレスを用いて終結するまで，単に割込みを禁止するだけでは十分でないことを指摘している．その理由は，他方のプロセスは他方の物理プロセッサ上で実行することができ，もし引渡されるアドレスを含むプロセスをもつ1つのセグメントを共用していれば，内側のプログラム・セグメントの実行中にそのアドレスを変更してしまうかも知れないからである．さらに Graham はその問題は呼ばれたセグメントと同一のリング内のデータ領域に引数リスト内のアドレスを複写し，そしてそのアドレスの正当性を確認すれば解決できることを示した．このようにして，呼ばれたプログラムと同一のアクセス権を持った他方のプロセス内にある1つのプログラムだけがその引き渡されるアドレスを変更できる．

その他にも引数を渡す問題がある．内側のリングにある1つのプログラム・セグメントがそれより外側のリングにあるプログラムを呼ぶ場合，内側のプログラムは外側のセグメントがアクセスできないそれ自身の内に引数を渡してほしいことがある．この問題は引数を外側のプログラムがアクセスできる領域に複写すれば解決できる．

リング・システムはハードウェアで有効な保護を十分利用できるが，正当性を確認したりどのリングが現在制御下にあり，特定のリング内にあるルーチンが終了した後にどのリングに制御を戻すかを記録するために依然としてソフトウェアの操作が必要である．ハードウェアに若干手を加えれば後で必要になる

雑用を減らすことができる．そのようなハードウェアの付加は Graham によって提案されている[20]．それは2.5節で述べたプログラム・ベース・レジスタにビットを付加して実行中のセグメントのリング番号を示すようにし，またセグメント・ディスクリプタとして格納されている情報にリング番号を付加している．この情報により実行中のセグメントがどのリングに属しているかを簡単に分かり，リングの境界を横切る場合を高速に処理できるようになる．現在のところ，この情報はファイル・ディレクトリに維持されている．

Graham によって提案されたこの機構はＧＥ-645 と同様なセグメンテーション・ハードウェアを持つ日本の計算機上にハードウェアで実現されている．この機械，HITAC 5020[33]，は各セグメント・ディスクリプタに2ビットのアクセス制御と4ビットのロックを持っている．ディスクリプタ・ベース・レジスタ内には4ビットのキーがある．これらの情報は次のように保護を行なう．先の議論のところで，キーとロックをリング番号として考えれば良いだろう．キーは実行中のセグメントに関連しており，ロックとの関係でそのプロセスの他のセグメントへのアクセスを管理する．もしそのキーが0（内側のリング）であれば，他のすべてのセグメントに対してどのようなアクセスも許される．もしそのキーの値がロックの値と等しいか，または小さいときは，そのアクセス制御（読出し専用，読出し／実行，読出し／書込み，すべて可）に一致した形のアクセスが許される．キーの値がロックの値より大きいときは，読出し専用アクセスは許されるが，書込みアクセスは決して許されない．キーとロックの値が等しくないセグメントに飛越しを行なおうとすると，割込みが起される．これはシステム全体でさらに経験が得られた時に期待できるハードウェアによる保護形式の一例である．

5.4.5 ケイパビリティ

タイムシェアリング・システムには保護を必要とする種々の形式の対象が含まれていると考えることができよう．その一例として，ファイル，プロセス，記憶装置のページあるいはセグメント，プロセス間のコミュニケーションのためのソフトウェア・チャネル，およびシステム動作あるいはシステム呼出しなどがある．それぞれの対象はその形式にもよるが，さまざまな量で割付けられ，別々の方法でアクセスされたり動作する．たとえば，ある人にはある長さまたはそれ以下の長さのファイルを生成することが許されたり，あるファイルを読出し専用としてアクセスが許されたり，プログラムとしてのみ実行可能，書込み専用あるいはこれらの組み合わせでアクセスが許されたりする．各々の対象に

ついてのこれら**オプション**（option）の性質および数は対象によって異なる．保護システムに要求されることは，種々の形式の対象を一様な方法で取扱い，プロセスに必要なだけの対象に対して最小限のオプションを持たせてアクセスを許すことである．この能力をもった方式は多層あるいは多レベルの保護を生み出す骨組となる．また，保護方式は実行のシステム・オーバヘッドが最小であることが望まれる．

先に示したように，保護を対象の上に置くものと，その対象へのアクセス経路上に置くものに考えることができる．対象へのアクセス経路上に置いた保護は対象の上に置いたものより一般にオーバーヘッドが少ないが，両方の保護形式がおそらく必要になろう．たとえば，1つのプロセスがある対象をくり返し参照する場合，そのプロセスにその参照を行なう権利が与えられていても，システムはその参照の度びにチェックする必要がある．XDS-940 システムでは，ある種のシステム呼出しはあるユーザのプロセスによってのみ行なうことができる．そのような呼出しのたびに，その要求を出しているプロセスを続行させてよいか否かを調べるために，権利を与えられたユーザのリストを検索しなければならない．この場合の保護は対象（システム呼出し）の上にある．プロセスに正当な対象へのアクセスを許すために，各対象への参照を線形アレー要素を通して行なう．たとえば，この要素の各エントリーをプロセスが生成されたとき，あるいは最初の参照が行なわれたときに初期化しておけば，チェックが最少になり，しかも保護が完全に行なえる．初期チェックの間，種々の対象に関連したテーブルを参照しなければならないことになるが，このような方式を対象へのアクセス経路上に置いた保護と呼ぶ．

種々の形式の対象を一様な方法で保護し，多レベルの保護を行ない，対象へのアクセス経路上に保護を置くという要求を満たして，ハードウェアおよびソフトウェアの対象を保護することについての考え方は，Dennis および Horn[31] の重要な論文で概説されており，また Lampson の貴重な貢献[34,35] によりさらにそれらが拡張され改良されている．この方式では，システム中のすべての対象は**ケイパビリティ**（capability）という名前が付られている．ケイパビリティは**ケイパビリティ・リスト**（capability list）あるいは**Cリスト**（C-list）に保持される．ケイパビリティは2つの機能を持つ．すなわち，(1) 対象に名前を付けること．すなわち，どの形式の対象が参照されており，それがどこに位置しているかを知らせること，および (2) このケイパビリティを使ってプロセスが使える対象へのアクセスに対するオプションを設定すること，である．Cリストは保護の範囲を定めており，Lampson はこれを**ドメイン**（domain）と呼

んでいる．Cリストは1つの対象と考えることができ，ケイパビリティによって保護される．ケイパビリティは1つの対象の保護された名前であるので，システムだけがそれを生成したり変更できる．ユーザ・プロセスにケイパビリティを持たせCリストを調べたり，その名前付きの対象に関連したオプションの数を減らすであろうが，Cリスト間で1つのケイパビリティを複写する機能を与えることができる．たとえば，あるプロセスが別のプロセスを生成するとき，それ自身の持つケイパビリティのサブセットでできたCリストをそのプロセスに組み合わすことができる．

もし下位のプロセスがそのCリスト中にない対象をアクセスしようとしたり，オプションにない方法でアクセスしようとすると，そのプロセスはブロックされ上位のプロセスに誤まり信号が渡される．その時，上位のプロセスは下位のプロセスを消滅させたり，そのプロセスのケイパビリティあるいはオプションを拡張するなどの適切な動作をとることができる．

プロセスが階層をなしているところでは，誤まりをそれに関連する2つの数[35]，すなわち**誤りクラス**（error class）と**誤り番号**（error number）によってより一般的に取扱うことができる．階層をなしているプロセス毎に**誤り選択マスク**（error selection mask）が関係付けられる．プロセスはそれに対する誤り選択マスクのビットが“1”である誤りクラスの誤りを受け付ける．誤りが生じると，それに対する誤り選択マスクのビットが“1”になっているプロセスが見つかるまで階層が調べられる．このプロセスは誤りクラスおよび誤り番号を引数として呼び出される．呼び出されたプロセスはその時必要な動作を行なえる．このケイパビリティの考えを実現する方法にいくつかある[31,34]．

各々のケイパビリティは1つの名前，すなわちその記憶場所を持つ．もし個々の記憶語にシステム以外は読出し専用というタグを付けるハードウェアがあれば，その場合のケイパビリティ名は記憶場所である．そのような機構もまた保護されるためにケイパビリティ名を持つことができる．ほとんどのシステムはこのようなハードウェアを持っていないので，ソフトウェア機構を使わなければならない．Dennis および Van Horn によって提案され，Lampson によって CAL システムに使われた機構は，Cリストを線形アレーとして構成し，ケイパビリティ名としてそのアレー内のインデックスを用いている．このようにして，もしあるプロセスが現在動作中のCリスト内にあるケイパビリティを示すと，指定された対象にアクセスする権利の証拠と解釈される．

CAL で実現されたケイパビリティは2語の60ビット語で表現されている．の語は対象の形式とオプションのビットを含んでいる．一方の語は対象の形式

とオプションのビットを含んでいる．2番目の語はケイパビリティに固有の名前と**主対象テーブル** (master object table)(MOT) へのポインタを含んでいる．またMOTは固有名とその対象へのポインタを持っている．この実現の方法は次のような利点がある．すなわち，(1) もしケイパビリティに損傷が加えられると，そのケイパビリティ内とMOTにある固有名は一致しないこと，(2) 対象を削除したり移動するときに，MOT 内のエントリーを変更する必要はなく，また修正をするためにすべてのCリストを検索する必要がないことである．

Lampson が提案した別の実現方法は，ケイパビリティを列で表わし，ドメインあるいはCリストを行で表わしたビット・テーブルを設定することである．このようにして，あるドメインはもしその行および適当な列のビットが“1”であれば，ケイパビリティを含むことになる．実際の対象へのマッピングはもちろん必要である．

さらに1つのことについて明確にする必要があろう．すなわち，Cリストに関連するものは何であろうかということである．多くの可能性がある．たとえば各プロセスは自分自身のCリストを持てるが，各々が自分自身のドメイン内で動作したり，多数のプロセスが同一のドメイン内で動作したり，また単一のプロセスのプログラムが同一あるいは別々のドメインで動作する場合がある．どれを選ぶかはもちろんその他の設計，あるいは実現上の判断に関係する．すべての場合について興味あることは，別々の保護領域の間で転送が起きた時に，何が生じるかということである．プロセスは潜在的に損害を与えないで1つのドメインから別のドメインへ勝手に移ることを許されるはずがない．Dennis および Van Horn による例はこの損害を与える可能性を説明している[31]．2つのプロセスAおよびBがルーチンSを使って別々の保護領域で動作しており，しかもSが別の保護領域にある場合を考えてみよう．AあるいはBの誤動作が他方のまちがった実行を生じさせないことを，またはSが使っている共通のデータ・ベースに損傷を与えるはずがないことを確かめたい．特に知りたいのは後のことである．もしAとBがSに任意の命令で入れるとすれば，Sは誤って共通のデータ・ベースを変更してしまうだろう．それゆえ，1つのドメインへの入口は制御しなければならない．それには保護**された入口点**すなわち**ゲート**を作りそれを通してドメインに入るようにする[31,34]．これらのゲートはケイパビリティで保護され得る対象である．特定のドメインで動作しているプロセスAは，Aにアクセスを許されたその他のドメインにあるプロセスによって複写され得るゲートに対してケイパビリティを生成することができる．そのゲ

ートから入ることによりAは正しく実行され，保護領域を変えることができる．

ドメイン間の飛越しは通常サブルーチン呼出しによって作られ，したがってなんらかの戻る方法を使う必要がある．Lampson は次のようなことを指摘した．すなわち，呼ばれたプロセスが戻るための別のゲートを生成することは望ましくない．なぜならば，呼ばれたプロセスはこのゲートを不適当なところで使うことができるからである．管理ルーチンだけがアクセスできるプッシュ・ダウン・スタックを使って戻りを格納すればこの問題は解決される（2.5.6節を参照）．そのエントリー内に戻り記憶場所と呼出しているプロセスあるいはドメインのCリストのポインタを退避する．

上の議論はケイパビリティの概念の基本的な考え方を概説している．その概念をさらに拡張し，発展させたものが Dennis, Van Horn，および Lampson によって議論されている[34,35]．その考え方について重要な点は，ケイパビリティは種々の形式の対象を一様な方法で保護し，多くのレベルの保護を作れること，またアクセスを許したプロセスおよび許されたアクセスの階級についてのリストを対象毎に保守し，検索する必要があるが，その対象へのアクセス経路上に保護を置くことができることである．

5.4.6 まとめ

本節で数レベルの保護が必要であることを指摘した．情報の保護はハードウェアおよびリソース割付けに使われる多数のシステム・テーブル中に存在する．融通性があり，一般的な Multics システムのリング構造の保護方式について概要を述べた．ケイパビリティ・システムはセグメント間あるいはリソース間のアクセス経路上に保護を置くソフトウェア機構である．保護についてはさらに次章で論じる．

参 考 文 献

1 Saltzer, J. H.: Traffic Control in a Multiplexed Computer System, MAC-TR-30, (thesis), Massachusetts Institute of Technology, Cambridge, Mass., July, 1966.

2 Lampson, B. W.: A Scheduling Philosophy for Multi-processing Systems, *Commun. ACM*, vol. 11, no. 5, May, 1968.

3 Coffman, E. G., and L. Kleinrock: Computer Scheduling Methods and Their Counter Measures, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, p. 11, 1968.

4 Schwartz, J. I., and C. Weissman: The SDC Time-sharing System Revisited, *Proc. ACM Natl. Conf. 1967*, pp. 262–271.

5 Fine, G. H., et al.: Dynamic Program Behavior under Paging, *Proc. ACM Natl. Conf. 1966*, p. 223.

6 Coffman, E. G., and L. C. Varian: Further Experimental Data on the Behavior of Programs in a Paging Environment, *Commun. ACM*, vol. 11, no. 7, p. 471, July, 1968.

7 Varian, L. C., and E. G. Coffman: An Empirical Study of the Behavior of Programs in a Paging Environment, *Proc. ACM Symp. Operating System Principles*, October, 1967.

8 Freibergs, I. F.: The Dynamic Behavior of Programs, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1163–1168, 1968.

9 Denning, Peter J.: The Working Set Model for Program Behavior, *Commun. ACM*, vol. 11, no. 5, May, 1968.

10 Nielson, N. R.: Computer Simulation of Computer System Performance, *Proc. ACM Natl. Conf. 1967*, pp. 581–590.

11 Nielson, N. R.: The Simulation of Time-sharing Systems, *Commun. ACM*, vol. 10, no. 7, pp. 397–412, July, 1967.

12 Randell, B., and C. J. Kuehner: Dynamic Storage Allocation Systems, *Commun. ACM*, vol. 11, no. 5, p. 297, May, 1968.

13 Randell, B., and C. Kuehner: Demand Paging in Perspective, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1011–1018, 1968.

14 Denning, Peter J.: Thrashing: Its Causes and Prevention, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 915–922, 1968.

15 Dijkstra, E. W.: Structure of The Multiprogramming System, *Commun. ACM*, vol. 11, no. 5, May, 1968.

16 Corbato, F. J.: A Paging Experiment with the Multics System, *MIT Project MAC Mem.* MAC-M-384, July 8, 1968.

17 Lauer, H. C.: Bulk Core in a 360/67 Time-sharing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 601–611, 1967.

18 Babcock, J. D.: A Brief Description of Privacy Measures in the RUSH Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 301–302, 1967.

19 Badger, G. F., Jr., and E. A. Johnson: The Pitt Time-sharing System for the IBM Systems 360, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 1–6, 1968.

20 Graham, R. M.: Protection in an Information Processing Utility, *Commun. ACM*, vol. 11, no. 5, p. 365, May, 1968.

21 Rehmann, Sandra L., and Sherbie G. Gangwere, Jr.: A Simulation Study of Resource Management in a Time-sharing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1411–1430, 1968.

22 Lampson, B. W., et al.: A User Machine in a Time-sharing System, *Proc. IEEE*, December, 1966.

23 Oppenheimer, G., and N. Weizer: Resource Management for a Medium Scale Time Sharing Operating System, *Commun. ACM,* vol. 11, no. 5, p. 313, May, 1968.

24 Reiter, Allen: A Resource-allocation Scheme for Multi-user Online Operation of a Small Computer, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* vol. 30, pp. 1–7, 1967.

25 Corbato, F. J., et al.: An Experimental Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* vol. 21, pp. 335–344, 1962.

26 Anderson, G. B., et al.: Design of a Time-sharing System Allowing Interactive Graphics, *Proc. ACM Natl. Conf. 1968,* pp. 1–6.

27 Fikes, Richard E., et al.: Steps toward a General-purpose Time-Sharing System Using Large Capacity Core Storage and TSS/360, *Proc. ACM Natl. Conf. 1968,* pp. 7–18.

28 Maurer, W. D.: An Improved Hash Code for Scatter Storage, *Commun. ACM,* vol. 11, no. 1, pp. 35–38, January, 1968.

29 Morris, Robert: Scatter Storage Techniques, *Commun. ACM,* vol. 11, no. 1, pp. 38–44, January, 1968.

30 Bobrow, D. G., and D. L. Murphy: Structure of a LISP System Using Two-level Storage, *Commun. ACM,* vol. 10, no. 3, pp. 155–159, March, 1967.

31 Dennis J. B., and E. C. Van Horn: Programming Semantics for Multiprogrammed Computation, *Commun. ACM,* vol. 9, no. 3, pp. 143–155, March, 1966.

32 Van Horn, E. C.: Computer Design for Asynchronously Reproducible Multiprocessing, *MIT Project MAC Tech. Rept.* MAC-TR-34, Ph.D. thesis, November, 1966.

33 Motobayashi, S., et al.: The HITAC 5020 Time Sharing System, *Proc. ACM Natl. Conf. 1969,* pp. 419–429.

34 Lampson, B. W.: Dynamic Protection Structures, *AFIPS Conf. Proc.,* vol. 35, *Fall Joint Computer Conf.,* 1969, pp. 27–38.

35 Lampson, B. W.: An Overview of the CAL Timesharing System, Computer Center, Univ. of Calif., Berkeley, 1969.

36 Spier, M. J., and E. I. Organick: The Multics Interprocess Communication Facility, *Second ACM Symposium on Operating Systems Principles,* Princeton Univ., Oct. 1969.

第6章

ファイル・システムおよび一般の入出力

6.1 ファイル・システム

6.1.1 概　　要

ファイルとは何か．ファイルとは関連した情報を集めて名前を付けたものである．ファイルは計算機システムにおいてビット，字，語などの要素の列として貯えられる．ファイルの例としては，プログラム，テキストの列，数値のアレーなどがある．ファイルはセグメントの定義においても与えられたすべての特徴を持っている．そして実際に Multics システムにおいては，語ファイルとセグメントは互換性を持っている．Multics システムはファイルを極めて一般的かつ統一的な方法で取り扱うように設計されている．XDS-940 システムは，統一的な方法ですべてのファイルを取り扱うわけではない．ユーザはシステムとより明白な形で交信するという煩わしさはあるが，ユーザがファイル・システムに要求する主な機能の多くを結局は提供している．

ファイル・システムの必要性は主記憶の大きさに制限があることによって起こる．この制限のために，ディスク，ドラム，テープ・ストリップ，紙テープ，磁気テープ，カード，および光学的記憶装置等の形をした補助記憶が必要となる．

ファイル・システムとは全体的な記憶管理システムの主に補助記憶を取り扱う部分である．ファイル・システムはアドレス付けや主記憶の割付けのために用いる方式とのインタフェースが円滑でなければならない．論理アドレス空間が非常に大きいMulticsのようなシステムにおいては，全体的な記憶管理システム自体がファイル・システムである．すなわち，Multics はプログラマがあたかも非常に大きな主記憶（論理アドレス空間）を使い，また補助記憶と主記憶間のすべての情報転送は，プログラマには見えないように設計されている．

Multics のユーザはファイル・システムを明白な手続きで扱うこともできる．一方，XDS-940 システムにおいては，そのユーザは常に明白な手続きでファイル・システムを取り扱わなければならない．

汎用タイムシェアリング・システムは次のような本質的な要求を満足しなければならない．

1. ユーザはファイルの作成，変更，および削除ができなければならない．
2. ユーザは互いの仕事を行うために，ある制御方法で互いのファイルをアクセスすることができなければならない．
3. ユーザはファイルにアクセスできる人を定めたり，またファイルの読出し，ファイルへの書込み，ファイルの内容の実行などのアクセスの形式についても制限することができなければならない．
4. ユーザは問題に適した形式で，ファイルを構成することができなければならない．
5. ユーザはファイル間で情報をコミュニケーションできなければならない．
6. ユーザはファイルが事故で削除されたり，壊されたりした場合のために，バックアップのファイルを準備することができなければならない．
7. ユーザはファイルを記号名でアクセスできなければならない．すなわち，ファイル・システムはユーザに関する限り，ハードウェアに無関係でなければならい．

汎用ファイル・システムに関する共通な技術的問題は，(1)十分な容量，アクセス速度，および利用できる補助記憶機器による経済的な記憶装置を得ること，(2)ユーザ間の適当なファイル保護ならびにシステム・ハードウェアおよびソフトウェアの誤動作に対する保護を達成すること，である．以下の節でこれらの必要性について論じ，その後他の形式の I/O について検討する．

一般に I/O システムの設計はファイル・システムとして概念的に統合することができる．この統合は，すべての I/O 機器を決ったシステム固有の名前を持ったファイルとして考えることによって達成される．そのシステムはそれらの名前をチェックし，それにしたがって情報を流すことができる．ファイルとしての I/O 機器と，より一般的なファイルのユーザに対する唯一の概念的な相違は，I/O 機器の多くは情報を単一方向にしか転送できないということである．ユーザは同一のシステム呼出し，ファイル・コマンド，および計算機言

語における出力文を，全ての情報転送演算を操作するために使うことができる．ファイル・システムを実際に実現する場合には，補助記憶と他のI/O機器では多くの相違が生じるが，それらについてここでは議論しない．

遠隔機器の操作を行なうソフトウェアは，種々の機器の取扱いに容易に適応でき，システムの他の部分と遠隔端末とのインタフェースを容易に適合できるように設計すべきである．最後にユーザとコミュニケーションするためのコマンド言語通訳ルーチンの機能を簡単に述べる．

ファイルおよび I/O システムは，システムの全プログラミングの大きな部分を占める．タスクを理解し管理させるために，その設計は機能モジュールに分けて行なわれる[6]．本章ではこのような機能分割について述べる．ファイル・システムやデータの管理に関する課題は非常に広い．本章は現在のタイムシェアリング・システムが個々の応用システムを作り出すための基礎として与えている基本的な機能についてのみ触れる．ファイル・システムのソフトウェア概念について議論する前に，ファイル・システムの設計に直接影響のある補助記憶機器のハードウェア的な特徴のいくつかを述べることは有用であろう．

6.1.2 補助記憶機器の特徴

タイムシェアリング・システムに使われる補助記憶機器は2つの主要な範ちゅう，すなわち**オンライン記憶**および**バックアップ記憶**に分けることができる．オンライン記憶機器は**直接アクセス**（direct access）または**ランダム・アクセス**（random access）機器とも呼ばれる．オンライン記憶機器には大容量磁心記憶，磁気ドラム，磁気テープがある．バックアップ記憶機器にはディスク・パック，データ・セル，磁気テープのように取りはずし可能な記憶媒体を持った機器がある．写真やレーザ技術を用いた機器は非常に容量は大きいが，情報を消すことはできない．どの機器にとっても重要な特徴は，そのアクセス・モード，アクセス時間，転送速度，容量，および価格である．以上に述べたすべての機器は，アクセスの方法が逐次アクセスである磁気テープを除いてはランダム・アクセスである．大容量磁心記憶については5.3.3節で述べた．そして1次ファイル記憶機器としての容量および低価格という条件をどちらも満たさないので，ここではこれ以上考察しない．他のランダム・アクセス機器は，語またはそれより小さな情報の単位に対してランダム・アクセスではないが，語や字情報のブロックに対してはランダム・アクセスである．逐次アクセスでテープ駆動時間の長い磁気テープは，オンライン・ファイル機器としては適さず，バックアップや特殊な応用に限定されるであろう．

アクセス時間や遅れ時間（アドレスされた情報が転送できるまでに待たなければならない平均時間）は，回転速度あるいは機器の機械的特性，読み書きセンサを位置付けるために使われる機構，および機器の大きさの関数である．アクセス時間は小型高速ドラム（容量は 10^6 字程度）の約4ミリ秒から，レーザや光学的記憶装置（容量は10^{10} 字程度）の1秒もしくはそれ以上といろいろである．これらの数字はオンラインとしての容量に対するものである．取りはずし可能な記憶媒体を持った機器は無限の容量をもつ効果がある．同じ価格差でもアクセス時間と容量の限界との関係でさまざまな機器がある．最もよく知られたファイル記憶機器は，数億字の容量をもち，約100～200ミリ秒のアクセス時間をもった大容量ディスク装置である．

転送速度と価格は記憶表面の情報記憶密度，読み書きセンサの数とそれに対応したアドレス，および制御電子回路の関数である．パッキング密度が増加すればする程，制御回路の価格は減少する．現時点においては，上に述べたディスク機器は1,000字あたり10セント，またはそれ以下で情報を貯えることができる．転送速度は現時点のディスクにおいては，1秒間に $100～200\times10^8$ 字であり，機器上のデータの並列的な読み書きによって主記憶の速度にまで増加できる．大きなタイムシェアリング・システムに必要な容量を確保するためには記憶装置の階層化が必要であろう．このような一般的な背景のもとで，直接アクセス機器の重要な空間割付けの問題に注目しよう．

6.1.3　直接アクセス機器のアクセスと割付け

概　要　ファイル・システムの設計における1つの目標は，システムのハードウェアをユーザに無関係にすることである．ユーザは**論理レコード**で仕事をする．システムの補助記憶およびI/O機器は**物理レコード**からより自然に構成されている．ファイルは1つ以上の論理レコードからできている．論理レコードは任意の長さや構造を持ち，それプロセスに依存している．物理レコードはそれぞれの機器に適した固定の長さをもつ．論理レコードは1つまたはそれ以上の物理レコードから構成することができる．もし論理レコードが物理レコードよりも小さいときは，I/O 転送や記憶効率を高めるために，普通1つ以上の論理レコードが1つの物理レコードにパックされる．磁気テープ・システムでは，物理レコードはそのテープ上に連続的に配列される．テープ・ファイルはそのファイル全体を書き換えずに付加，または削除して変更を行なうことが難しいという欠点がある．直接アクセス機器の特徴は残りの部分を変更することなしにファイルの一部を簡単に修正することができることである．ファイル

の更新に有効なこの融通性を持つ一方，直接アクセス機器は要求された情報を即座にアクセスできるという特徴がある．このような利点があるこれらの機器における記憶空間の構成は種々の方法で取扱うことができる．ここで，含まれている概念に対する感覚を得るために，いくつかの手法を述べる．議論はディスク装置に対するものであるが，その概念はドラムやデータ・セルのような他の機器に対しても同じように容易に拡張できる．スワッピング・ドラムおよびディスク・システムに対する割付けやアクセス要求の最適化を自動的に行なう特別なマイクロプログラムされたプロセッサの例は6.1.3節で論じる．

直接アクセス機器の物理的構造　磁気ディスクは磁気材料でコーティングしたプレートを何枚か積み重ねたものからできている．そのプレートは一定速度で回転し，**トラック**（track）と呼ばれる同心円の集まりになっている．情報はそのトラック上に記録される．そのトラックもまた一般に**セクタ**（sector）と呼ばれる部分に分割されている．大低のシステムにおいて，転送可能な最小情報量がセクタである．その構造を 図 6.1 に示す．

ディスク装置には一般的な2つの形式がある．1つは各トラックに付随して一個の読み書きヘッドを持っており，特定のトラックへのアクセスが電子的な速度でできるようになっているものと，他は1個以上のヘッド／アームの組合せ機構を持ち，心要なトラックに機械的に位置付けをしなければならないものである．この位置付けの動作を**シーク**（seek）と呼ぶ．シークを実行するために必要な時間は，ヘッドおよびアームの数やディスクの大きさによって異なるが，約75ミリ秒からそれ以上の範囲にわたる．ディスクに関するアドレス情報に応じて，制御装置はプレート，トラック，およびセクタ・アドレスを送らなければならない．情報のファイルはセクタまたはセクタのグループと同じ大きさの物理レコード単位でディスクの上に構成される．物理レコードはディスク上のどこにでも書込める．しかしながら，図 6.1 に示すように，プレート当り1個のアームを持ち，読み書きセンサが櫛状に組合されたアームに据え付けられているようなシステムの場合は，転送およびシークの効率を考えて，連続したセクタまたは並列プレート上の同一トラックに物理レコードを記録するようにする．この櫛状のユニットは1つにまとまった位置決め機構として動作する．

2つの重要な問題を解決する必要がある．すなわち，(1) ディスク上の物理レコードの記憶場所と，適切なシークエンスを記憶することと，(2) 新しいファイルと古いファイルの拡張のために使うことのできる未使用空間を記憶することである．何人かのユーザが利用可能な全空間を使いきってしまわないよう

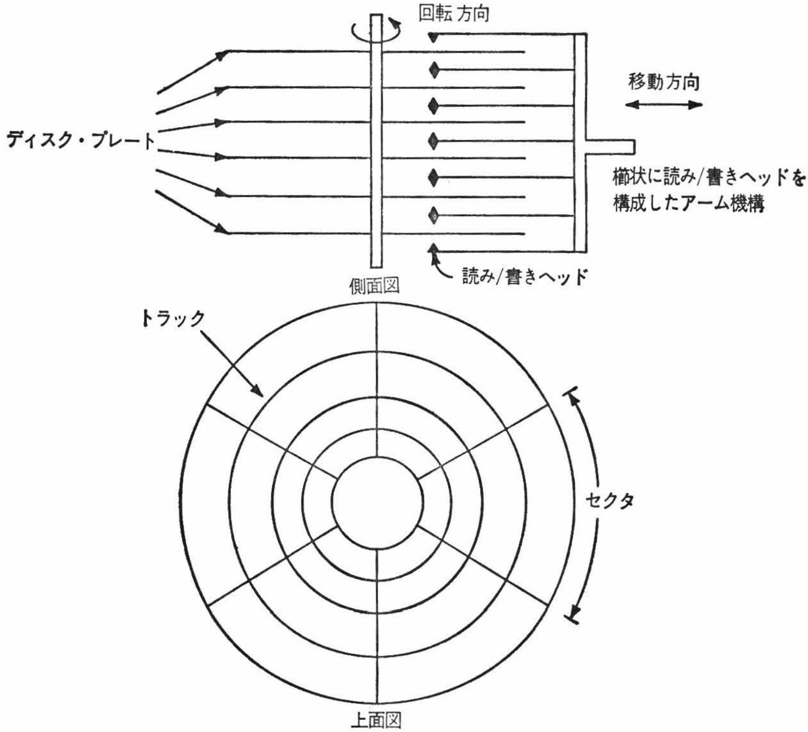


図 6.1　ディスク装置の典型的な構造

にするために，別の機器が使えない限り，個々のユーザが取り得る空間に関して制限を設ける必要がある．

基本ファイル・システムは，ファイルが内部的にどのように構成されているかを知る方法を持たない．ファイル・システムにとっては，ファイルは情報ユニットの系列にすぎない．ファイルの内部的構造は応用プログラムあるいは特別なシステム・データ管理ルーチンの責任である[15]．応用が異なれば，ファイルのアクセス法も違ってくる．基本ファイル・システムがうまく設計されていると，ユーザには必要なアクセスの融通性が与えられる．直接アクセス機器の実際的な割付けについて述べる前に，アクセス法に関して要求される2つの一般的な事柄について概要を述べることにする．

逐次およびランダム・アクセス　最も単純なアクセス形式は**逐次アクセス**

(sequential access) である．逐次アクセスでは，ファイルは論理レコードと同じ大きさの連続したブロック単位にアクセスされる．多くのプログラムは I/O を逐次ファイルとして扱っている．逐次ファイルは磁気テープに似ている．論理レコードをランダムにアクセスしたり書込んだりはできない．すなわち，次にアクセスあるいは書込まれるレコードは，直前にアクセスあるいは書込まれたレコードに続いていなければならない．システムは物理ブロックを読出し，論理レコードを要求しているプロセスに転送する．出力の際には，システムは機器に書出すために，論理レコードを寄せ集めて物理レコードに組合わす．逐次ファイルは，ランダム・アクセス記憶の機能を完全には利用しないが，そのような機器上に作ることができる．XDS-940 システムでは，逐次ファイルに対して次の字，語，または一定の語のブロックを読み書きするためのコマンドを用意している[10]．プロセスとファイル間の転送のために，字を語にパック，あるいはアンパックすることもシステムによって行なわれる．XDS-940 では，ユーザ・プロセスが論理レコードの長さに関する情報を維持しなければならない．

ランダム・アクセス記憶装置は**ランダム・アクセス**法によって効率的に利用される．ランダム・ファイルを用いて，ユーザはファイルの開始点，転送する語，字，あるいはビット数に関連した記憶場所を指定することにより，ファイル中のどの論理レコードもアクセスできる．さらに改善すれば，ユーザはランダムに指定した開始点から逐次的にファイルをアクセスすることもできる．これらの技法を変形して，種々のアクセス法が作れる．

空間の割付け　ファイルがどこに格納されているかを記録するために使われる記憶空間，利用可能な未使用空間，および 1 つのファイルをアクセスするのに要する時間（ディスクのアクセス回数で測られる）などの間に種々のつり合い問題がある．ファイルを可変長にするためには，その中に含まれている物理レコードを互いに**つなぎ合わせ** (chain) なければならない．このつぎ合わせはランダム・アクセス補助記憶構成の基本概念となっている．ファイルの物理レコードをつなぎ合わせるために XDS-940 で使われている技法は，**インデックス・ブロック** (index block)[10] と呼ばれる．インデックス・ブロックは連続した語のテーブルにすぎない．テーブルの i 番目は，ファイル内の i 番目の物理レコードのアドレスを持っている．ファイルは，おそらく，いくつかのインデックス・ブロックが必要な大きさであろう．インデックス・ブロック内の 1 語を使い，次のインデックス・ブロックのアドレスを含ませることによっ

て，ブロックをつなぎ合わせることができる．あるいは，いくつかのレベルのインデックス・ブロックを使うこともできる．インデックス・ブロックはディスク上に置かれ，通常その長さは1セクタである．先頭のインデックス・ブロックのアドレスは，**ファイル・ディレクトリ**（file directory）（6.1.5節参照）と呼ばれるテーブル内にある．この概念を 図 6.2 に示す．

物理レコードは有効な情報の最後を示す記号を含むことができる．上に述べた技法は多くのシステムで使われているが，少しばかり変更されている．他に物理ブロックをつなぎ合わせる方法は互いのポインタを1つのテーブル内にまとめて持つのではなく，物理ブロック自身の中に持たせることである．インデックス・ブロック法はシステムが管理するために，ファイルがどこにあるかについての情報をたった1回，あるいは数回のアクセスで得ることができ，各物

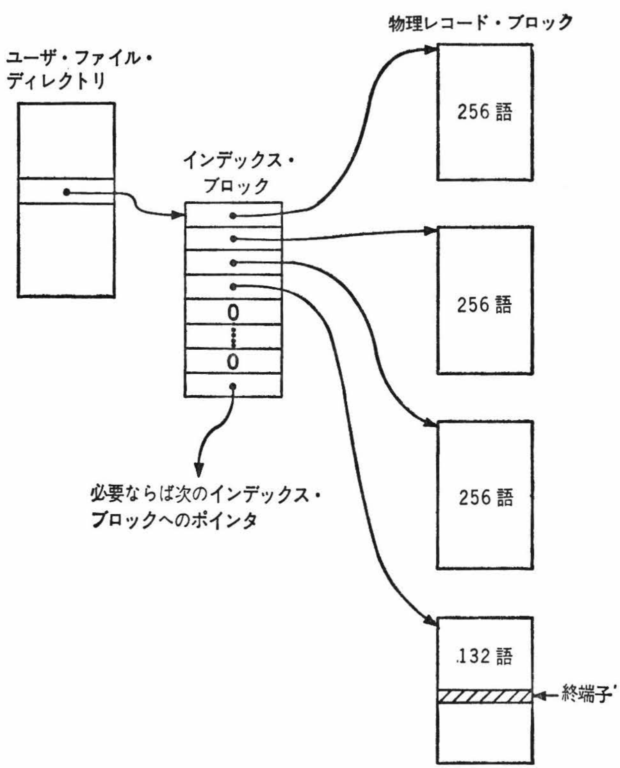


図 6.2　ファイルにおけるデータ構成

理レコードをアクセスする必要がないという利点がある．管理上の仕事には，データをファイルに挿入する必要がある場合に，ファイルの中に物理ブロックを付け加えたり，もはや不必要となったブロックを未使用空間に戻したり，多重ブロック・ファイルにランダム・アクセスのためのブロックを配置したり，ある種の誤動作の後でファイル・システムの確かさを調べること，などが含まれる．先に述べた構成をとるファイルは，逐次的にもランダムにもアクセスできる．逐次アクセスの場合，システムは読み書きすべき次の物理レコードに関する情報を記憶している．

これらの概念を示す例を **図 6.2** に示すが，そこでは物理ブロックが256語の長さで，アクセスされるファイルが900語であるとしている．さらに，ユーザはファイルの先頭から500語以後の30語をアクセスし，それをユーザのアドレス空間に特定のセル単位に格納したいとする．システムはこのファイルに対するインデックス・ブロックのアドレスを知るために，ユーザのファイル・ディレクトリを調べ，ディスクをアクセスしてそのインデックス・ブロックを読み出す．システムは500から529までの相対ファイル・アドレスが2番目と3番目の物理レコードに存在することを決定でき，インデックス・ブロックからこれらの物理レコードのアドレスを読み出すことができる．そこで，システムはバッファ中に2番目の物理レコードを読み込み，500から511までのファイル語の内容をユーザの記憶空間の適当な場所に移す．システムはさらに3番目のブロックをアクセスし，512から529までのファイル語をユーザの記憶空間の適当な場所に移す．この時ユーザはアクセスするファイルの名前と必要情報に関するファイル内の相対記憶場所を知っているだけでよく，ファイルが格納されている機器の種類や，ファイルがどのようにして，あるいはどこに格納されているかについては何も知らなくてよい．

もしユーザが相対記憶場所212から始まる25語を変更したい時は，システムが相対記憶場所216から236までの中にその情報を重ね書きする．

レコードを重ね書きするよりはむしろランダムに挿入するようなファイル・システムを作るには，相対記憶場所によってではなく，レコード名によってファイルをアドレスする方法を用いることが必要である．このアクセス法は論理レコード名，あるいはその数，レコードが始まる物理ブロックのアドレス，および論理レコードの長さを含むインデックス・テーブルを必要とする．もし論理レコードが物理ブロックの境界をまたがる場合は，物理ブロック中のポインタが次の物理ブロックのアドレスを示すか，あるいはこの情報をインデックス・テーブルの中に入れる．上に示したような相対記憶場所に対するランダム・

アドレス方式を使ったファイル・システムは，レコード名によってアクセスできる形式，あるいは別のテーブルを使った種々の構造に変更可能である．

非常に大規模なランダム・ファイル，あるいは逐次ファイルを作ることができるが，物理空間は書き込まれた情報を持ったファイルのアドレス空間を含む物理ブロックに対してのみ割付けられることが必要である．すなわち，システムは情報が書き込まれた時に物理レコードを割付け，インデックス・ブロックを更新することができる．

ファイルの削除中に物理ブロックにおいて生じる隙は後で述べるファイル・バックアップや再ロード操作中に取り除くことができる．システムは再ロード中にブロック転送を能率的に行なうために，各々の物理ブロックを隣合わせにすることもできる．その時，ファイル・システムはディスクへのアクセス回数を減らすことによって，ブロック転送速度を上げるために，適切な割付けを行なうことができる．このようにして，ユーザは早い応答を得ることができ，システムはより効果的にディスクを活用することができる．

未使用空間の記録　上に述べたことは，ファイルに必要な空間を作り出すために物理レコードをつなぎ合わすという重要な概念を導入している．**未使用空間**を記録しておくという問題はまだ検討を続けるべきものである．未使用空間を記録するには種々の方法があるが，ディスク・ファイルの大きさや未使用セクタを探すのにどのくらい時間をかけてよいかなどに依存する．1つの方法は未使用セクタをつなぎ合わせる方法である．この方法は簡単で，主記憶のわずかな空間しか必要としないが，もしシステムが1個以上のセクタを要求すれば，多くのアクセス時間が必要となる．他の方法は**ビット・マップ**(bit map)（テーブル）を使うことである．ビット・マップにおいては，各セクタ毎に利用できる1ビットで，十分大きな語の集まりを取り扱うことができる[10]．対応するセクタが未使用であればそのビットがセットされ，使用中ならばクリアされる．トラック当り8セクタからなる32トラックのディスク・プレートに対するビット・テーブルを 図 6.3 に示す．

プレート毎にさらに多くのトラックを持ち，かつ何枚かのプレートを持つディスク装置にとっては，特別の計算機語長をもつ構造のビット・テーブルとなり，いくらか複雑になる．ビット・マップ中のビット位置（語インデックスおよび語内のビット位置）を与えれば，簡単な計算でディスク上のセクタの対応する物理アドレスを決めることができる．この方法は限られた記憶容量の機器ではうまくいくが，大容量記憶機器を利用する場合にはビット・マップのため

セクタ＼トラック	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1				1						1					1	1			1			1					1		1			
2	1		1	1			1				1			1	1	1		1				1					1		1			
3		1		1		1		1				1		1	1	1		1	1	1	1	1			1					1	1	
4			1			1	1		1	1	1	1			1			1		1				1	1	1	1			1	1	1
5			1		1			1		1		1	1	1	1	1		1	1	1	1	1		1			1			1		1
6		1				1				1		1				1					1		1	1					1	1		
7			1						1			1					1	1		1	1		1			1				1	1	1
8			1	1		1		1								1	1	1		1	1					1		1	1	1	1	

図 6.3 32トラック8セクタのディスク・プレートに対する未使用空間ビット・マップ

に非常に大きな主記憶が必要となる．この問題に対する1つの解決法は，主記憶内にはビット・マップの一部だけを保持しておき，ビット・マップの残りの部分は補助記憶に格納しておくことである．ビット・マップのための記憶領域を確保することは簡単であり，この方法を使えば経済的になる．主記憶内にあるマップの一部で示される記憶領域がすべて割当てられてしまった時（マップのすべての語が0になる），マップの別の部分が主記憶に持ち込まれる．記憶領域を未使用に戻すためには，マップ中の適切なビットを変更するために，解放される機器の記憶領域を表わすマップの部分を主記憶に持ってくる必要がある．このビット・マップとのやり取りは，戻されるセクタのリストを作ることにより減らすことができ，この場合マップの部分が主記憶に持ち込まれる時に更新する．

未使用空間管理のもう1つの方法は，各ユーザに主記憶中の連続したブロックをあらかじめ割付ける方法である．バックアップ動作後の機器の再ロードは，ユーザ・ファイルを連続したブロックに再パックすることになる．この方法では，次の未使用物理レコードの位置とレコードがいくつ使われたかのカウントを記録するために1語が使われる．戻されたブロックは未使用というマークが付けられる．しかしながら，システムにとっては別の語でその空場所を記録しない限り，次に再パックするまでは本質的には失われたものとなる．ユーザの物理ブロックが全部使われてしまうと，同様な方法であふれ領域に空間を割当てることができる．この割付け方法もやはりつなぎ合わせの技法を使っている．この方法による未使用空間の記録は高速に行なえ，主記憶をほんの少ししか使わない．この方法はファイル記憶領域の周期的なダンプと再パック操作に依存している．もしこの再パック操作が通常のバックアップ・プログラムとして取り扱えるならば，その動作にはあまり時間がかからない．

マイクロプログラム記憶制御装置および割付け装置　4.3.3 節で理解したように，システム中で頻繁に使われ，実時間の要求があると判断され，しかもそこに含まれる問題を明確に理解している部分は，マイクロプログラム化した制御装置に組み込んでしまうよい例である．そのような例の1つに補助記憶の制御と割付けがある．特に高速ドラムやディスクのようなスワッピングに使われる補助記憶の場合である．ここでどのような概念が問題になるかを示すために，マイクロプログラム制御装置によって実現できる高速スワッピング用ディスク・システムの割付けおよびアクセス法に関して，いくつかの特長を概説する．各トラック毎にヘッドを持ち，セクタの大きさがページの大きさに等しいドラムについて考える．このようなドラム上の未使用空間は，一般にビット・テーブルに記録される．現存のシステムでは，機器との間の転送を最適化するため，セクタの順序にしたがって要求の待ち行列を作り，ドラムと主記憶間の情報の流れをできるだけ一定に保っている．ビット・テーブルや待ち行列を維持することは，システムにとって大きな重荷となり，それらの機能を小型のマイクロプログラム化したプロセッサに任せた方が有利である．この小型プロセッサを**ドラム・プロセッサ**と呼ぶことにする．

ドラム・プロセッサには次のような特徴が要求される．読み書きセンサの下にある現在の，あるいは次のセクタ番号を格納する高速レジスタが必要である．このレジスタは，ドラムからのタイミング・パルスによって駆動されるカウンタになる．ドラムに対するページ書き込み要求は，転送すべき主記憶のブロック番号とともにドラム・プロセッサへ送られる．ドラム・プロセッサはこの要求を書き込み待ち行列に入れる．ドラム・プロセッサは，前節で述べたように，未使用空間のビット・テーブルをそのローカル記憶に保持している（あるいは主記憶をアクセスする）．

ビット・テーブルは，利用可能な空間を探索するために用いるマイクロ操作がヘッド下の次のセクタ番号を保持するレジスタを介してビット・テーブルをアドレスするように作られる．このマイクロ操作は最初の利用可能なセクタのアドレスを戻して，それが書き込みヘッドの下にくるようにする．そのセクタが利用可能であることを示すビットは使用不可を示すようにセットされ，その後転送が開始される．書き込まれたページのディスク・アドレスは中央プロセッサに戻され，再びそのページを主記憶に読み出す時に使われる．言い換えると，CPU がページの書込み操作の準備に使われる時間は，書き込み要求を出す時間とページが書き込まれたディスク・アドレスを格納する時間だけであ

る．ドラム・プロセッサは常にドラムの位置を知っており，次に有効なドラム上のブロックを利用できるので，ドラムへの書き込みは極めて少ない待ち時間で処理される．ページより大きなブロックを取扱うドラム・プロセッサを作ることもできる．その場合，ドラム・プロセッサはレコードの開始アドレスをCPU に戻しながら，ドラムの物理レコードの割付けとつなぎ合わせを行なう．

読み出しの際は，CPU はドラム・プロセッサに読み出し要求，転送するページあるいはブロックの主記憶アドレス，およびドラム・アドレスを送る．その時，ドラム・プロセッサはドラム・セクタに従ってローカル記憶内に読み出し要求の待ち行列を作る．ドラム・プロセッサはドラムの位置を常に知っているので，主記憶とドラム間の情報の流れをできるだけ一定に保つような順序で転送を行なう．読み出し後に，そのドラム空間は利用可能としてビット・テーブルに印される．しかしながら，この方法は5.3.2節で述べた“書込まれていないページと書き込まれたページの論理”が使えない．そのような技法のために種々の考え方が検討できよう．

ここにあげた概念は，マイクロプログラム・プロセッサを使う可能性を秘めたシステムの種々や，特殊化したマイクロプログラム・プロセッサの設計に移行しそうな考えの種類のみを示すために紹介したものである．

まとめ　本節でユーザは論理レコードで仕事をするのに対し，オンライン補助記憶機器は物理レコード単位に最も自然に割付けられることを知った．物理レコードはバッファリングに都合の良い大きさの情報のブロック，あるいは機器上のアドレス可能な最小単位に関係したものである．オンライン機器は逐次的にもランダムにもアクセスできる．

融通性をもって記憶の割付けを行うために，物理レコードは論理レコードやファイルを形成するように互いにつなぎ合わせる．つなぎ合わせのためのポインタは物理レコード内，あるいは**インデックス・ブロック**と呼ばれるポインタのテーブル内に保持される．通常，インデックス・ブロックは物理レコードの長さであり，補助記憶内に格納される．インデックス・ブロックへのポインタは**ファイル・ディレクトリ**と呼ばれるテーブル内に保持される．

未使用空間を記録することも考える必要がある．この目的のためにビット・テーブルを使用することを示した．大容量機器ではビット・テーブルは非常に大きくなる．そこでビット・テーブルは補助記憶内に保持し，部分的に主記憶に入れることになる．大型機器を割付ける別の方法は，各ユーザに対してあらかじめ大きな連続ブロックを割付けることである．未使用空間の記録はこの方

法によって容易になる．ファイルが削除されると，空き空間が作り出される．空き空間によって示される未使用空間は機器を再パックするまでシステムに属さないままにしたり，空き空間を記録するためのつなぎ合わせやその他の技法を使うことができる．これらの概念をマイクロプログラム・プロセッサの設計を概説することによって示した．このプロセッサは補助記憶機器に関する記憶空間の割付け，およびアクセスのスケジュールを自動的に行なうものである．

6.1.4　ファイルの共用とアクセス法

概　要　XDS-940 や Multics システムにおいてファイルの共用やアクセス法がどのように行なわれているかを調べると，これらの機能に関連した多くの概念が明らかになる．設計目標の違いにより，これら2つのシステムはファイルの共用およびユーザ相互にファイルのアクセスを許すという問題に対して別々の方式をとっている．第2章で知ったように，Multics は全てのファイルをセグメントとして取扱い，全てのセグメントは決った方法で取扱われる．すなわち，全てのセグメントは完全な一般性を失なうことなく共用でき，システム中に存在するのはたった1つの複写でよい．XDS-940 システムの場合，設計者達は使用頻度と使用形態を基にして異なったファイルの類別があり，これらを別々に扱うとシステムの経済性が達成できると了解していた．

MULTICS における共用とアクセス法　第2章で Multics におけるファイル（セグメント）のアクセスおよび共用の方法に関係した多くの機構について述べた．6.1.5 節では別の題材を提供しよう．本節ではユーザから見た特徴のいくつかを簡単に述べる．

Multics システムにおいてファイルはセグメントであり，それゆえ，システム中のいかなるファイルも名前と記憶場所によって直接的にアドレスされる．6.1.5 節で述べる**ファイル・ディレクトリ**と呼ばれるテーブルを使って，システムはファイルを捜し出す．第2章で述べたように，セグメントをアドレスしているユーザにそのアクセスが許されると，ユーザにはセグメントは公開されたものになる．Multics システムは，ファイルを明白な形でも暗黙の形でもアドレスできる．オペランドをアドレスする全ての機械命令は，Multics システム中のいかなるファイル（セグメント）をも暗黙の形でアクセスできる．Multics のユーザは自分が名前を付けた新しいセグメントに出力することによって，明白な形でファイルを作り出すことができる．ユーザはシステムに明白な形のコマンドを与えてファイルを削除する．Multics における全てのプログラムはリ

エントラントに書かれるので，完全に共用でき，システム中にはたった1つの複写が存在するだけである．Multics ではすべてのコンパイラ，あるいは他のサブシステムはセグメントであり，その他のシステムやユーザのセグメントと同じように取扱われる．データを構造化するために，たとえアドレス・ポインタを含んでいたとしても，データ・セグメントは完全に共用できる[3]．そのような共通のデータ・ベースのたった1つの複写がシステムには必要であり，ユーザの論理空間では何ら位置の違いによる問題は生じない．

Multics のファイル・システムの設計目標は次のことを考えれば自然にでてくる．すなわち，計算機の構造に無関係な大きな仮想記憶を備え，プログラミングの高度な一般性（名前だけを知っていればプログラムが使える機能で，記憶装置への要求やそのプログラムが呼び出す他のプログラムなどを知っている必要はない）を与えることにより，かつまた，プログラムのたった1つの複写ならびに正当と認められたデータの共用を許すことによって，大きなユーザの社会から出てくるさまざまな計算の要求に報いるのが Multics の総括的な設計目標[3] である．

開発中のソフトウェアおよびハードウェア技術のいかなる点においても，これらの非常に価値ある本質的な目標を達成するにあたって遭遇する技術問題を解決するには，ハードウェアの価格，システムの複雑性，オーバーヘッド，および開発期間などの点において代償を支払うことになる．これらの価格は，大きな記憶空間を必要とする新しい問題に取り組むことのできる可能性，応用プログラムやシステムを容易により経済的に開発できること，十分にプログラムやデータを共用することによってより効果的な共同作業および過去の労力を利用できること，そして一般的に非常に簡単に利用できること，という利益を比べて測られなければならない．経験によると[17]，計算機を設置するあたっての投資と運営上の経費の大部分は，応用プログラムの開発および保守に費やされる．Multics ファイル・システムの1つの設計目標はこれらの重大な出資を減らすことにある．

XDS-940 における共用とアクセス法　XDS-940 の設計者達には Multics の設計者達よりもさらに限定された目標があった．そのため，使用頻度ならびに価格の点からシステムの複雑性を増したり，開発期間を長くかけても利益にならないと思われるところは一般性を犠牲にした．彼等の目標の1つは，高度の会話形式で限られた数の端末をサポートするシステムを開発することであった[16]．各プロセスは一般に 16K より小さい．設計者達はシステムの設計概念に

関して実験を行うことを望み，そのために融通性を必要とした．以下に述べるつり合い関係は，中規模のシステムに関連して出てきたものであることに留意してほしい．

XDS-940 システムには3種のファイルがある：

1. システム中にただ1つの複写だけがあって，多くのユーザに共用されるファイル．この種のファイルは，コンパイラ，テキスト・エディタ，デバッグ・ルーチン，およびアセンブラのようなプログラムであり，**サブシステム**と呼ばれる．現在のところ，データ・ファイルのたった1つの複写を共用できる機能はない．
2. 個々の複写を得ることによって，全てのユーザがアクセスできるファイル．これらのファイルは**システム・ライブラリ・ファイル**と呼ばれる．
3. ユーザにとって半ば私有的なファイル．

上のそれぞれの種類のファイルは少し異なった方法でアクセスされる．

サブシステムは XDS-940 システムに接続された端末からのコマンドによってアクセスされるリエントラント・プログラムである．システムはサブシステムを読み出し，ユーザのプログラム記憶テーブルに空間を割当て，サブシステムの擬似リラベリング・レジスタを準備し，そしてサブシステムに制御を移す．サブシステムはそこで必要とする別のファイル名を要求する．あるコンパイラが記号ソース・コードの入力を含むファイル名とコンパイルされた2進コードを出力するファイル名を要求するのはこの一例である．ある特定のユーザに対して一度にただ1つのサブシステムだけが実行される．サブシステムとして実行するプログラムの性質を考えるとそれほど重大とは思えないこの制限は，有効な価格と一般性とのつり合いから出てきたものの1つである．

ライブラリ・ファイル機構によって，ユーザは互いの仕事を確立し，他のユーザが開発した多くのプログラムを同時に使うことができる．払われるべき犠牲は，ある特定プログラムの多数の複写が主記憶中に入れられるという可能性である．設計者達は与えられた 940 のライブラリ・ファイルがいくつかのプロセスによって同時に使われる頻度は低く，それゆえに完全な一般性を持ったただ1つの複写を共用する機構を導入することは，考えられる価格に引き合わないと思った．ある特定のシステム導入に際しての仕事の性質によって，もし与えられたライブラリ・ルーチンがいくつかのプロセスによって同時に使われるならば，サブシステム中にこれらのルーチンを作っておけばよい．そのような

変更を行うために必要な労力は，そのような可能性を考慮してプログラムが作られたかどうかに依存する．与えられたライブラリ・ルーチンを使うプロセスはすべてそれ自身の複写を持っているので，1つのプロセスが多くのライブラリ・ルーチンを使っても論理空間中の位置決めには何ら問題は生じない．したがって，ライブラリ・ルーチンはプロセスの論理空間中ならばどこにでもロードされ得る．

ユーザは許されていれば他のユーザの半ば私有のファイルをアクセスすることができるが，それは端末またはサブルーチンのどちらかを介してユーザのファイル空間に必要なファイルを複写するようにシステムにコマンドを出すことによって行なわれる．その結果，システム中にはそのファイルの2つの複写が存在することになる．そのかわりユーザ側でのわずか余分の労力と2次記憶空間をいくらか使うことによって，システム開発の容易さが得られた．システム中の情報の多数の複写を使うことに関連していくつかの潜在的な危険性が存在する．もし1つのファイルの多くの複写があるときは，プログラムから誤まりを取り除くような場合に，1つの複写を更新するために他の全ての複写を捜し出す必要がある．1つのファイルの多数の複写があり，これらのうちのどれもがオリジナルと置き換えられるならば，他のユーザに知られずに修正された複写は新しいオリジナルになり得る．

940 システムの設計者達はこれらの要因を考慮し，小さなユーザ集団であれば，多数の複写の共用は適当に制御できると考えた．ファイルの共用が広く普及されれば，そのファイルをサブシステムあるいはライブラリ・ファイルにすることができる．その場合でもシステム内に多数の2進プログラムの複写が残ることになる．ここで再び，価格と一般性とのつり合いをどうするかが決定された．

データ・ファイルのたった1つの複写を共用する場合をこれから述べる必要がある．XDS-940 システムはこのための機構を備えていない．データ・ファイルはライブラリあるいは半私有ファイル機構を通じてのみ複数のユーザによってアクセスされ得る．第2章で見たように，もしデータがアドレスを含んでいなければ，ページングはデータのただ1つの複写を共用することに制限を与えない．データ構造を作るためのアドレス・ポインタがデータの中に入っていれば，そのような共用されるデータはそれをアクセスする各プロセス・マップ中の同じ位置に置かれなければならない．

ファイルの共用に関連した問題を論じてきたが，ここではユーザがどのようにファイル間でコミュニケーションするかを簡単に示す．Multics のユーザはファイル間のコミュニケーションを次の方法で直接的に行なえる．すなわち，

通常のロード，ストアの機械命令でデータ・ファイルをアクセスするか，I/O命令を使うか，あるいは通常の飛越し機械命令でプログラム・ファイル間を飛越すことによって行なう．リンケージ処理については2.5.6節で述べた．6.2.4節で述べたように，XDS-940 システムのユーザはシステム呼出しを用いて必要なデータ・ファイルを明白な形で**オープン**（open）しなければならず，それから特別のシステム・プログラム化されたオペレータでもってファイルの入出力をしなければならない．この特別のシステム・プログラム化されたオペレータはプログラムにとってロードやストア命令のように簡単に使えるが，システムの時間を若干必要とする．システムは主記憶と補助記憶間のバッファリングと実際の転送を取扱う．

XDS-940 システムでは逐次アクセスとランダム・アクセスの2通りのアクセス法が利用できる．逐次アクセスに有効なシステム呼出しを用いれば，ユーザは自分の記憶空間とファイル上の次の順番にある位置との間で字，語，あるいはブロックを読み書きすることができる．ランダム・アクセスに有効なシステム呼出しを用いれば，ファイル内のいかなる場所でも語あるいはブロックを読み書きできる．逐次アクセス法は高レベル言語の文で使われる通常の方法である．ランダム・アクセス法は高レベル言語で使うにはサブルーチン・パッケージが必要である．

プログラム・ファイル間のコミュニケーションを行なうために，XDS-940 のユーザは中間データ・ファイルを作り，システムに制御を移してから端末からのコマンドあるいは先に述べたシステム呼出しに基づいたサブルーチン・パッケージを使って，新しいプログラム・ファイルに制御を移すために明白の形で要求を出さねばならない．したがって，XDS-940 システムにおいて支払う代償は，ユーザは今何が起っているかをより多く知る必要があることと，ユーザの端末からシステムにより頻繁に交信しなければならないことである．

XDS-940 システムの方法は，多くの応用に極めて妥当であることがそのシステムでの色々な経験からわかっているが，より一般的なシステムを要求するような構造化されたデータ・ファイルを必要とする応用もたくさんある．

この議論は2つの考えられる設計方法における相違を示している．この2つの方法の類似点に関して2，3の有益な事柄がある．Multics におけるファイルも XDS-940 におけるファイルも名前を持った一連の情報ユニットである．両システムともファイルの内部構造はユーザに任せている．また両システムともファイルの先頭に対する相対的な位置を指定することによって，そのファイル内をランダムにアドレスできる．また両システムはプログラムやデータ・フ

ァイルを共用させるという本質的な問題に対する解決をも与えている．

まとめ　本章と第2章での議論は，ファイルを共用することに関する多くの重要な概念を導き，仮想プロセッサのアドレス空間の欠くことのできない一部分として取扱われる大規模システム用のファイル・システムと，仮想プロセッサのアドレス空間外にアドレス空間が存在するとして取扱われる中規模システム用のファイル・システムとの類似点や相違点を示した．前者の方法はユーザがシステムとより頻繁に交信する必要があり，仮想プロセッサとファイルとの2つのアドレス空間で仕事をしなければならない．次にファイル・システム構造の性質とファイル・ディレクトリの概念について考えてみる．

6.1.5 ファイル・システムの構造

ファイル・ディレクトリ　6.1.1 節でファイルはシステムに対するただ1つの情報ブロックであると一般に考えられることを示した．個々のファイルはシステムから見ると何らの構造も持たない．しかしながら，ファイルを基本ユニットとして見なせるような構造に作ることは可能でありまた有益である．この概念を以下に述べる．Multics システムにみられるようなより一般的な形でこの問題を考える前に，XDS-940 のようなシステムを引合いに出して，いくつかの基本概念を導入することは有益である．

最初の話題はユーザのファイルの記録と各ファイルに関するアクセス制御および管理上の情報を維持することについてである．基本概念は**ファイル・ディレクトリ**の概念である．各ユーザのファイル・ディレクトリはそのユーザが次にそのファイルを使うまでディスクあるいはその他の記憶媒体上に固定的に保持されている．そしてユーザがファイルを使う際には，そのディレクトリはシステムによって主記憶内に持ち込まれる．システムは各ユーザのファイル・ディレクトリの記憶場所を持つ**主**（master）あるいは**ルート**（root）ディレクトリを保持しなければならない．そのディレクトリもファイルと見なされる．

ディレクトルは各ファイル毎に次のような情報を持っており，それらは一般に連続した語からなるテーブルとして格納されている：

1. 文字列によるファイル名，あるいはそのような文字列を示すポインタ
2. ファイルの形式，すなわち2進，字，あるいは特定のシステムで定義された他の形式
3. アクセス保護（多くはファイルをアクセスする他のユーザのリストに

対するポインタ，および読出し専用，書込み専用，読出し／書込み，あるいは実行専用のようなユーザに許されるアクセスの種類)

4. ファイルが格納されている場所を直接示す情報，あるいはそのような情報を含むデータに対するポインタ，すなわち6.1.3節で述べたインデックス・ブロックに対するポインタ

5. ファイルを定義した日付け，使用頻度，あるいはシステムの設置やシステム設計者に便利と思われる付加的情報など

ユーザは自分のディレクトリをアクセスしてファイル名を変更し，自分の端末にファイル名を印字させ，また他のユーザがアクセスする権利の追加，変更，削除ができる．またファイルを削除したり，ある特定のファイルに対するファイル形式を調べることができる．1.5.5節で述べたように，ユーザは課金番号とその他の識別で定義された抽象的概念であることを思い出してほしい．ユーザは1人の人物であり得るし，同じ課金番号を持つプロジェクトのような人の集団でもあり得る．たとえば，もしユーザが課金番号とプログラム番号という2つの要素で定義されていれば，同じ番号を持つすべてのユーザはある1つのユーザの集まりを構成する．ファイル保護に関する問題は6.1.6節でさらに述べる．

ファイルの形式　2つの一般的な**ファイルの形式**に**固定**（permanent）**ファイル**と**一時**（temporary）**ファイル**がある．固定ファイルはその所有者が明白な形で削除するまでシステム内に存在する．これに対し，一時ファイルはある特定の端末が動作中の間だけ存在する．一時ファイルはユーザがログ・アウトする時にシステムへのコマンドによってそのファイルを固定ファイルに変換しなければ削除される．

一般的なファイル形式にはさらに**字**ファイルと**2進**ファイルの2つがある．字ファイルは端末のキーボード上の字を符号化した連続記号を含む．この字ファイルは語長にしたがって語毎にいくつかの字をパックする．このパック操作とアンパック操作については6.1.3節で述べた．2進ファイルはデータまたは実行するために直接主記憶にロードできる情報を含む．2進ファイルであるか字ファイルであるかはシステムにとっては重要ではない．ファイル・ディレクト内の指示はユーザにとっては注意書きとしてのみ意味を持つ．ファイル・システムはファイルをビット系列として考える必要がある．

単純なファイル・システムの構造　単純なファイル・システムの構造は，

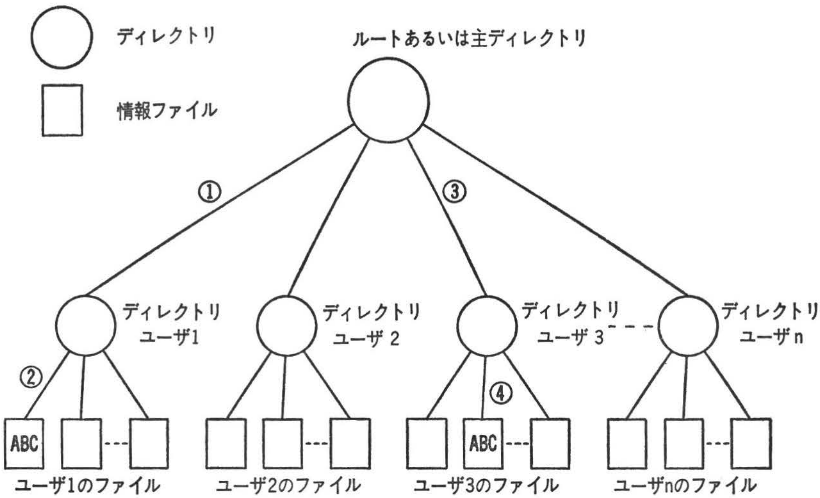


図 6.4 単純な 2 レベル・ファイル・システム構造

図 6.4 に示すように，2つのレベルを持ったツリー構造をしている．XDS-940 のファイル・システムはこの形式をとっている．図中，円はディレクトリ・ファイルを表わし，四角は情報ファイルを表わす．数字は説明に使う識別子である．ファイル・ディレクトリ中の各エントリーは名前を持ち，枝で表わされる．たとえば，1のエントリー名は USER 1 であり，2のエントリー名は ABC というように，ある特定のシステムにおけるファイル命名慣習にしたがった字系列である．4のエントリー名も ABC ということもある．ファイル名に関する唯一の制限は，あるファイル・ディレクトリの中には同じファイル名が2つ以上あってはならないということである．ファイルの**システム名**（system name）はルート・ディレクトリからそのファイルに至るエントリー名を連続したものである．たとえば，枝2と枝4が示すファイルのシステム名は，それぞれ USER 1：ABC と USER 3：ABC である．システム名（ツリーを伝わってファイルまでの経路をたどるので**ツリー名**（tree name）と呼ばれる）は，各ファイルに固有のものである．

システムは主ディレクトリ中のユーザ・ディレクトリのエントリー名を暗黙のうちにわかっており，誰のプロセスが実行中であるかをも知っているので，ユーザは自分のファイルを参照する時はユーザのファイル名を使うだけでよい．もしあるユーザが他のユーザのファイルをアクセスしたければ，完全なツ

リー名を宣言しなければならない．たとえば，USER 1がXという名前の自分のファイルにABCという名前のファイルを複写したいときは，次のような形式のコマンドを出さなければならない．

COPY USER 3 : ABC TO X

システムはUSER 3のファイル・ディレクトリを検索し，USER 1がファイルABCを複写することが許されているかどうかを調べる．もし許されていれば，ファイルは複写される．

より一般的なファイル・システムの構造　2レベルのツリー構造は非常に有効であるが，より一般的な構造のファイルを使えば，情報システムの設計や管理が簡単になる状況を思い浮べることができるであろう．たとえば，ユーザに属するファイル・ディレクトリではなく，プロジェクトあるいはサブプロジェクト毎に任意の深さにファイルをまとめたいこともあるし，またドキュメント検索システムのようにツリー状の構造をもつ情報を格納したいことがある．

Multicsシステムで達成されたように，2レベル構造は一般化できる．すなわち，図6.5に示すように，ディレクトリが任意の深さの情報ファイルのみならず，他のディレクトリ・ファイルをも指示できるようになっている．以下に述べるMulticsに関する題材はDelayとNeumann[1]の論文に基づいている．

先に述べたように，円はディレクトリ・ファイルで四角は情報ファイルを示す．先に述べたすべての命名法がこのより一般的な場合でも用いられている．たとえば，1のファイルはツリー名がA : H : E : Fである．明白に宣言しなければ，ツリー名はルートに対して相対的に与えられる．ユーザは特定の時点で**作業ディレクトリ**（working directory）と呼ばれる1つのディレクトリを使用していると考える．システムは約束にしたがって作業ディレクトリを割付けることができ，またユーザは引数にそのディレクトリに対するツリー名を持つコマンドをシステムに出して，どのディレクトリを自分の作業ディレクトリとするかを明白に示すことができる．たとえば，今述べた一般性を持ったファイル・システムを構成する方法を図6.6に示す．このシステムはライブラリ・サブツリー，ユーザ・サブツリー，システム・サブツリーという3つの主要なファイルのサブツリーを含んでいる．ライブラリ・サブツリーは数学的なファイルを示すサブディレクトリを持っている．したがって，統計ファイル，マトリクスを取扱うファイル，微分方程式を解くファイル等に対するサブディレクトリを持つことになる．またライブラリ機能の他の分類のための構造が用い

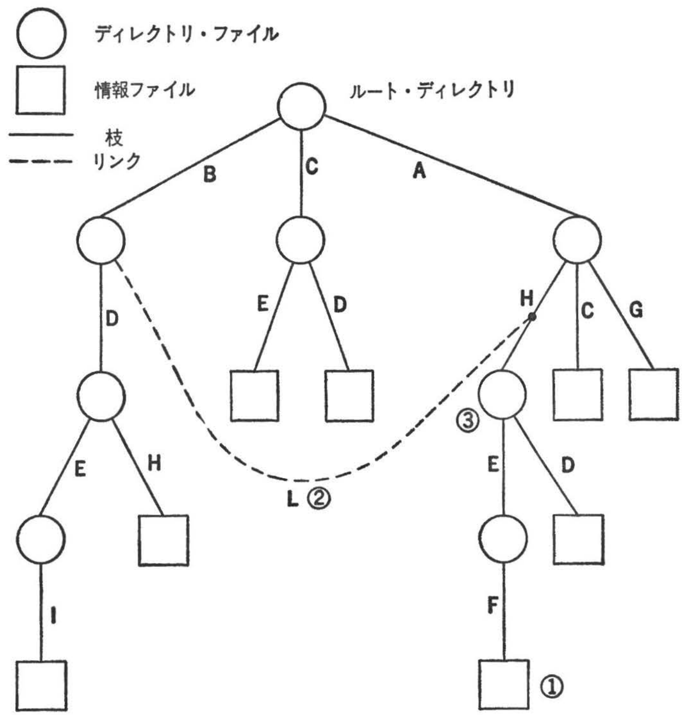


図 6.5 一般的なツリー構造のファイル・システム

られることもある．システム・サブツリーは機能や保護権利などに基づいたルーチンを色々分類するサブツリーを持つ．

ユーザ・サブツリーは各ユーザに対するディレクトリのためのエントリーを持っている．そのエントリーの各々は既に自分自身の情報ファイルとディレクトリ・ファイルのサブツリーを持っている．明白な形で仕事を指示しない限り，システムは各ユーザに作業ディレクトリとして主ディレクトリを割付ける．アクセス制御に関するすべての記述がディレクトリに関するアクセス制御を含めてこれより一般的な場合にも適用される．Multics システムでは現在の作業ディレクトリに関連して相対的にツリー名を指定することができる[1].

Multics ではセグメントが現在実行中のプロセスに対して未知の外部セグメントを参照する場合は，2.5 節で述べたように，システムはファイル記憶階層を探してディレクトリのエントリー中に外部セグメント名を見つけ，その名前をそのプロセスに対して公開しなければならない．ファイル名はあるディレク

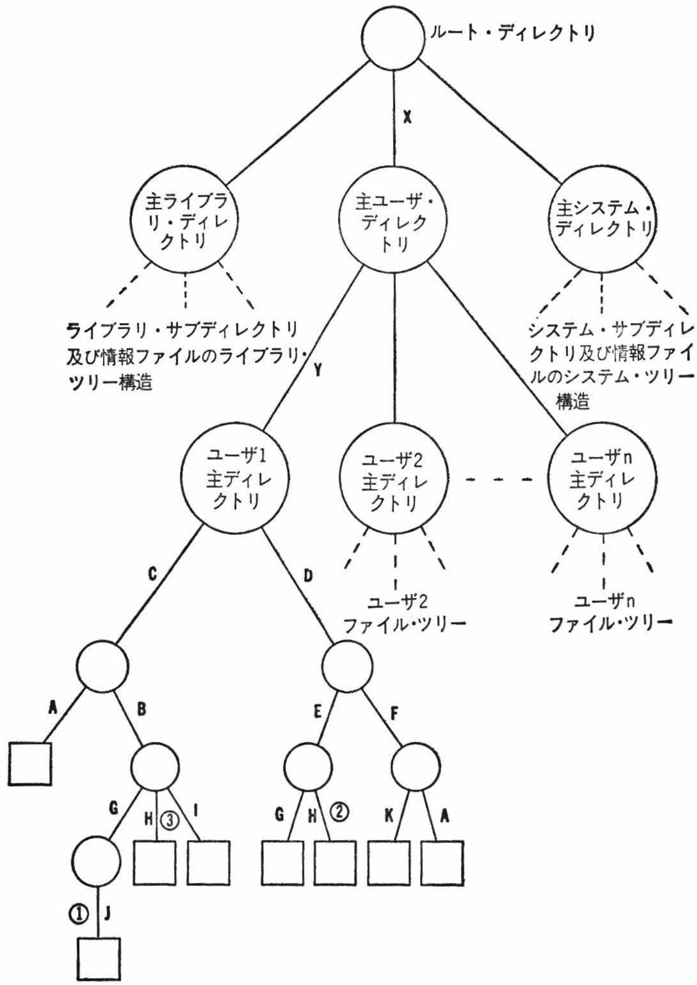


図 6.6　特定なファイル・システムの例

トリに対しては固有のものであり，特定の名前を持ったファイルがいくつかシステム内に存在する可能性があるので，ユーザは必要なファイルを見つけ出すために通常の探索法によるか，あるいはユーザ自身の探索法をシステムに与えなければならない．たとえば図6.6を参照すると，ユーザ1のプロセスが経路名X：Y：C：B：G：Jを持つセグメントJで実行し，その時点でそのプロ

セスにとっては未知のHという名前を持つセグメントを外部参照すると仮定する．ファイル構造の探索法によって，システムは3で示すセグメントHか2で示すセグメントHをプロセスに割当てる．もし次に述べる探索法を用いれば，3で示すファイルHが見つけ出される：

1. 参照しているファイルのすぐ上位のディレクトから探索を始める．
2. ツリーに沿って登る前にディレクトリの下にあるサブツリーを探す．
3. 次に上位のディレクトリで定義される各々のサブツリーを探す．

ユーザの主ディレクトリから始め，右から左へと探索していけば，2で示されるセグメントHが見つかる．またユーザは探索のための指針として経路名を与ることもある．システムはこの経路名で示される最も低いディレクトリから探索を始め，ルートとなっているディレクトリを探す．もしその名前のセグメントが見つからなければ，指定された探索経路の次に高いディレクトリへ移り，そのディレクトリで定義されるサブツリーを探索する．これが繰り返される．主ユーザ・ディレクトリにまで探索が進むとアクセスを許されている他のユーザ・ディレクトリを1つ1つ探索するか，あるいは慣例にしたがってライブリ構造を調べる．ここで述べたことの要点は，複雑なツリー構造の中で相互にセグメントを参照するユーザは，探索法に対する認識が必要であるというとである．

Multics におけるもう1つ別の概念である**リンク**（link）について手短かに説明する．リンクはファイル・ディレクトリであるが，図6.5に点線で示すように，ファイルを指すのではなく，他のディレクトリのエントリーを指すものである．リンクは名前を持つことはできるが，リンクが指す枝にアクセス制御があるので，リンクに関連したアクセス制御はない．リンクは階層構造における他の場所の枝への近道として働き，リンクが指すエントリーによって明記されるファイルを直接指す枝としての効果を持つ．たとえば図6.5で，リンクLは3で示すディレクトリを直接指すディレクトリBのエントリーを作るという効果を持つ．ツリー名の概念は今や**経路名**（path name）というものに一般化された．経路名はツリー名であるか，あるいはすべての枝の名前およびルートまたは作業ディレクトリから指定されたファイルまでのリンク名を連続して並べたものである．たとえば，3で示されるファイルの経路名はA：HあるいはB：Lである．

まとめ　ファイルに関する基本的な情報，すなわちファイル名，記憶場

所，アクセス制御，およびその他の管理上の情報はファイル・ディレクトリと呼ばれるテーブル内に保持されることが分った．ファイル・ディレクトリ中に保持されるファイルの記憶場所は一般にインデックス・ブロックに対するポインタである．ファイル・ディレクトリは補助記憶内に格納される．

単純なファイル・システムの構造は各ユーザのファイル・ディレクトリの記憶場所を持つルート・ディレクトリから成る．ユーザやシステムに情報システムを構成するためのより大きな融通性を与えるには，もっと一般的なファイル・システムの構造が便利である．ディレクトリをファイルとして扱うことにより，一般的なファイル・システムの構造が得られる．これはツリー構造である．ファイル名は定義されるファイル・ディレクトリの中では個有のものである．固有のシステム名はそのツリーのルート・ディレクトリから定義するディレクトリまで進む時に通過するディレクトリ名を連続してファイル名にしたものである．ファイルをシステム名ではなくユーザ特有の名前で参照する場合は，その探索の方法を示さなければならない．

6.1.6　ファイルのアクセス制御

ユーザ社会でファイルを共用できる能力はタイムシェアリング・システムの重要かつ本質的な目標である．オンライン記憶装置をより容易にアクセスできることは人々が仕事を行う上で特別の刺激を与える．ファイルを共用するためにはファイルのアクセス制御システムが種々のアクセス法を用意している必要がある[1,13]．匿名の遠隔端末から中央の計算機装置にますます情報が集中するにつれ，記憶された情報の秘密を守り保護する危険のない方法を用意するという大きな社会的責任がシステム設計者にかかってきている[14]．本節では今日まで開発されたファイル保護の考えを論じる．この重要な問題はさらに研究および開発が進められる必要がある．理想的なアクセス制御システムは，おそらくそのシステムを実現する人達によっても傷つけられることのないシステムであろう．

システム設計によって異なるが，ユーザが定義できるアクセス権は，誰でもファイルをアクセスできることを意味する**自由アクセス**（free access）と特定のユーザだけがアクセスを許されることを意味する**制限アクセス**（restricted access）がある．アクセスを与えられたユーザは読出しだけができたり，書込みや修正，付加することができたり，実行のためロードすることができる権限を持つ．これらのモードは**利用属性**（usage attribute）と呼ばれる．ユーザ毎に違ったアクセス権と利用属性が許される．たとえば，あるユーザには関係す

るプロジェクトにおいて何人かの共用作業者がおり，彼等には削除または変更できる完全な権限の利用属性でファイルのアクセスを許し，その他の人には読出しや実行のみの利用属性でファイルをアクセスできるようにする場合がある．これらの共同作業者を明確に列挙する空間をとるかわりに，課金番号がある種のユーザのためのアクセス権を決める基本となることがある．通常，ファイルの所有者，すなわち，ファイルが最初に作成される時にログ・インで使われた特定の識別子を持つ人だけがアクセスの特権や利用属性を追加，変更，削除することができる．

Multics システムでは TRAP[1] と呼ばれる強力なアクセス属性の実現が提案された．特定のユーザに対して TRAP 属性が与えられると，ユーザがファイルを参照する時にそのファイルのアクセス制御テーブルで指定されたプログラムに飛越し（トラップ）が起こる．この機構によって，ファイルの所有者は誰が自分のファイルを参照しており，どの位の頻度で参照されていかに関する情報を常に得ることができる．アクセス特権と利用属性はアクセスを許すユーザのリストを作り，種々のアクセス属性をビットに符号化することによって簡単に実現できる．論理的にはアクセス制御情報はファイルの先につながる枝（アクセス経路）にあるものと考えられる．

この種の制御に加えて他のモードの保護も述べる必要があろう．同時アクセスが許される場合，あるユーザがファイルを読出している間に他のユーザがそのファイルを変更しないことを保証するためにはインタロックが必要である．1人以上のユーザがファイルを読み出している時には読み出しインタロックがかかっている．これによっていかなるユーザもファイルに書き込めないようになる．同様に誰かがファイルに書き込みを行なっている時にも書き込みインタロックがかかっており，他のユーザが読み書きできないようになっている．

これまでに述べたファイル保護に関して要約し，さらに詳しく見ていくことにする．一般に必要となるファイルの保護機能は次の通りである[1,18]：

1. 偽って他のユーザになります者に対するファイル保護．この問題に対する保護はユーザがシステムにその識別子を提示するログ・イン時に取扱われる．安全性の特別処置を講じるために，識別子には印字されない文字が含まれることがある．
2. ファイルの共用を許されていないユーザによるアクセスに対する保護．この問題は一般にどのユーザあるいはどの種類のユーザがファイルをアクセスできるかを示すアクセス制御属性を組み合わせることによっ

て解決される．アクセスを制限するには次の3つの主な方法がある．

a　ファイル名を秘密に保つ．多くの場合これは困難であり，この方法を用いているシステムはしばしばファイル名を変更することが必要となる．ファイル名の一部として印刷されない合い言葉を割当てれば，この方法を改善できる．

b　ファイルの所有者，システムの全ユーザ，あるいは課金番号のような一般識別要素を基にしたユーザ層に対してアクセスを制限する．ユーザは自分のファイルの各々に対してアクセスできるユーザ層を指定することができる．この方法の欠点は1人のユーザにファイルのアクセスを許せば，そのユーザ層のすべてのユーザにアクセスを許さない訳にはいかないことである．ユーザ層による制限は実現するのに容易であるという利点がある．

c　アクセスできる個々のユーザを指定する．この一般的な機能を用意するにはもっと強力なソフトウェアが必要である．この方法は，一般にファイルの所有者がシステム・コマンドで変更できるリストによって実現できる．Multics では各枝にアクセス制御リストと呼ばれるそのようなリストを持っている．

3.　与えられた権限以外の方法によるファイルのアクセスに対する保護．権限のあるユーザまたは許されるアクセスの種類を示しているユーザ層毎に利用属性を定義することによって，この問題は解決される．たとえば，利用属性には読み出し，読み書き，追加，実行などがある．どのシステムもすべてのモードを持つものではない．読み出しと書き込みという属性はほとんどのシステムに存在する．ユーザ層毎のアクセスの場合，アクセス・モードはその層のすべてのユーザに対して同一である．

4.　ファイルの所有者がファイルを取扱う際に起こす誤りに対する保護．上に述べたアクセス・モードの属性はファイルの所有者自身が起こす誤りを防ぐために使うことができる．次節で述べるバックアップ機構はハードウェアおよびソフトウェア誤りに対する保護のみならず，過失によるファイルの削除や自分で与えた損害に対しても保護する．

6.1.7　ファイルのバックアップと回復

概　要　　システム設計者はハードウェアおよびソフトウェアが誤動作を起こし得ることを仮定しておかなければならず，設計の初期段階から信頼性，バックアップ機構，回復手順について考慮すべきである．本節ではファイル・シ

ステム用の保護機構について述べ，第7章ではシステムの他の観点からこの問題を論じる．

システム内のファイルは時間と労力を莫大に費したものであり，それゆえ非常に重要なものである．中核となるファイルを失なうと会社が確産することもあるし，研究プロジェクトが数ヵ月遅れることもある．誤動作の原因にはCPUや主記憶の一時的な誤動作から機器上のデータ・トラックが読めなくなったり，読み書きセンサが記録面を傷つけ壊してしまった場合のように機器全体の破損に至るまでさまざまである．

周期的ダンプ　バックアップ手順のうちで最も一般的なものはディスク・パックあるいは磁気テープのような可動媒体上にすべてのファイルおよびファイル・ディレクトリを周期的にダンプすることである．誤りが生じた時，そのファイルはダンプ・テープから再ロードできる．この最も単純な形式のファイル保護機構にはいくつかの欠点がある：

1. 通常，ダンプを行なう間システムを中断する必要がある．
2. ダンプ操作は大きなファイル・システムでは非常に時間がかかり，数十分から時には数時間にわたることもある．
3. 回復は最新のダンプから行なわれる．したがってダンプ操作は1日に1度行なうのが普通であり，ユーザは数時間分の仕事を失なうことになる．

ダンプ操作においては，ユーザが記憶装置の連続した領域に再ロードするためにユーザ・ファイルを集めてひとまとめにするというある種の操作を同時に行なうことができる．そのような再ロードを行なうことによりアクセス速度を増し，システムの能率を上げることができる．

インクリメンタル・ダンプ　上に述べた問題を回避するより手の込んだダンプ手順に**インクリメンタル・ダンプ操作** (incremental dumping)[1,18] と呼ばれるものがある．この方法を使うと，システム使用中に変更されたファイルの複写はユーザがログ・アウトする時にダンプ・テープ上に複写される．システムは使用中に変更された各ファイルのファイル・ディレクトリのエントリーにビットをセットする．ログ・アウト時にあるルーチンが変更されたファイル名とそのファイル・ディレクトリに対するポインタをダンプすべきタスクの待ち行列に加え，優先順位の低いシステム・プロセスがそのファイルをダンプで

きるようにする.

この機構でもユーザが長時間端末を使っている場合には危険が残る．しかしながら，低い優先順位のダンプ・プロセスがファイル・ディレクトリを連続的に走査し，変更されたファイルを見つけ出すようにすれば，この方法は改善できる．そのようなファイルが発見された時，そのファイルのファイル・ディレクトリのエントリーと一緒にファイルがダンプされる．ファイルが変更されたことを示す符号はその時リセットされる．ファイル名や個々のユーザのアクセス権を変更するようなディレクトリ・エントリーに対する変更もまた重要であり，したがって保存する価値がある．この場合，すべてのファイルを保存する必要はないが，ファイル・ディレクトリのエントリーは保存される．ダンプ・プロセスの優先度を適当に調整することによって，各ディレクトリを走査する平均時間を調節できる．このインクリメンタル・ダンプ法はMulticsで使われており，もちろん多量のデータを生み出している.

ダンプ情報による回復　システムの故障を回復するにはすべてのインクリメンタル・ダンプ・テープを長時間かかって調べ，各ファイルの最新版を探索しなければならない．したがって，システムの破滅的な故障をより速く回復できるようにするために，Multics システムでは**システム・チェックポイント・ダンプ**[1] と呼ばれるダンプを周期的に行なう．これはシステム・ファイルとファイル・ディレクトリの複写を行なうものである．回復するために，最新のシステム・チェックポイント・ダンプが最新のインクリメンタル・ダンプとともに再ロードされる．この処理でより速くシステムを動作状態にもどすことができ，インクリメンタル・ダンプの再ロードが低い優先順位で行なわれる間，多くのユーザのサービスができる．すべてのユーザ・ファイルをダンプするユーザ・チェックポイント・ダンプは2，3週おきに行われる．そして最新のユーザ・チェックポイント・ダンプはそれより古いインクリメンタル・ダンプが見つけ出される前に再ロードできるので，安定した古いファイルを持ったユーザのための再起動を早めることができる．この手順による回復の過程をMulticsシステムのプログラム・マニュアルと同様な図で 図 6.7 に示す．ユーザ・チェックポイント・ダンプが時刻ＴＵに行なわれ，最新のシステム・チェックポイント・ダンプが時刻ＴＳで行なわれ，重大な故障が時刻ＴＦで起こったとする．システムがユーザにサービスできる点までシステムを回復するためには，(1) システム内に各ユーザの最新ファイルを置くために時刻ＴＳにさかのぼってインクリメンタル・ダンプ・テープを再ロードする．(2) そこで最新のシステム・チ

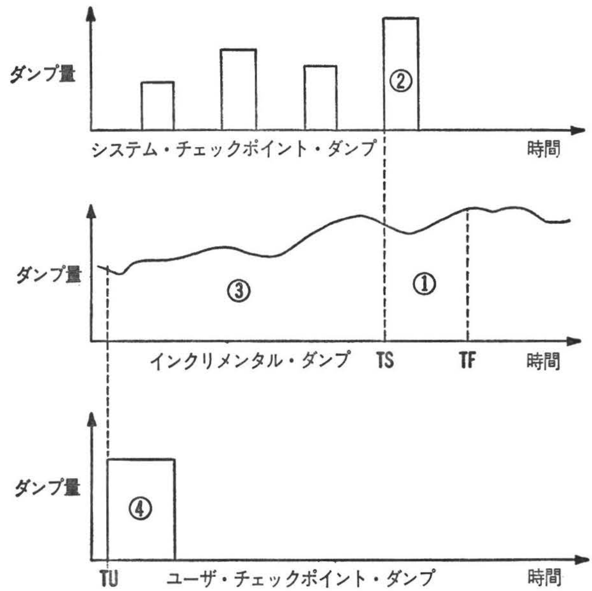


図 6.7 種々のダンプを用いた回復手順

ェックポイント・ダンプを再ロードする．この時点で新しいファイルの作成をするために待っているユーザはシステムを使えるようになる．(3) ここまでくると，時刻TUにさかのぼってインクリメンタル・ダンプをバックグラウンド・タスクとしてロードできる．(4) その後に最新のユーザ・チェックポイント・ダンプの再ロードを続ける．この時点でファイル・ディレクトリが更新され，ファイルのごく一部を除いてすべてのファイルが再格納されたことになる．ダンプ・テープをさらにさかのぼって調べるのはおそらく意味のない努力であろう．再格納されていないファイルのディレクトリ・エントリーは，そのファイルが要求される時にシステムがインクリメンタル・ダンプ・テープをさらに前に戻って探索しなければならないことを指示している．

プロセッサまたは記憶装置の誤りからの回復　前節での議論は破滅的な故障およびたまたまファイルを削除してしまったユーザがシステムに対して最新のインクリメンタル・ダンプまたはその他のダンプから対応するファイルの再ロードを要求する場合に当てはまる．大部分の故障は破滅的なものではなく，故障が生じた時点で更新されていたファイルあるいは主記憶内のファイル・デ

ィレクトリで示されていたファイルだけに影響を与える．このような故障が生じた時には，主記憶中のファイル・ディレクトリのエントリーは誤ったアドレスを指すか，あるいはどこか矛盾がある可能性がある．こういった故障から回復するには，すべてのファイルを再ロードする必要はない[5,13]．主記憶中のファイル・ディレクトリに矛盾がないかを調べれば回復できる．このファイル・ディレクトリを調べる手順の詳細はシステムに大きく依存して異なる．ほとんどのシステムは暗黙または明白な形でこの検査に便利な冗長情報を持っている．システム・プログラマはこのような矛盾がないことを調べる検査をするにあたって，自分のシステムについての知識を駆使しなければならない．この検査と回復過程によって，少なくともシステムを再始動した時に損害が広がらない点まで誤りを回復できる．損害を被った（あるいは被っていると心配される）わずかなファイルに対しては最新のインクメンタル・ダンプ・テープを再ロードし，ユーザが認識できるようにファイル・デイレクトリにフラグをたてるか，あるいは単にユーザが認識するためのフラグをたてるかのどちらでもできる．後者の場合，ユーザはそのファイルが実際に損害を被っているかどうかを調べることができ，その後特別に再ロードを要求できる．

その他の形式の誤りには補助記憶装置上の使用可能な空間を記録するために使っているテーブルを壊すことがある．この場合できることはこれ以上損害が生じないように情報の矛盾を取り除くことである．そしてチェックポイント・ダンプを行なうような時点で，ファイルを退避しながらすべてのインデックス・ブロックの構造を調べ，どの物理レコードが使用中でありまたどこが末使用であるかを再調査することである．

まとめ　誤りからの保護や回復は可能ではあるが，そのような手順の詳細な設計は簡単ではなく，設計者に創作力が要求される．周期的なファイルのダンプ，およびシステム・テーブル，ファイル・ディレクトリ，インデックス・ブロック等の構造の無矛盾性の検査という少なくとも2段階の保護が必要である．設計者は起りうる種々の故障やそれらがシステムに与える影響について明確に想定できなければならない．設計過程でこれらの問題を考える時点が早ければ早い程，おそらく問題を軽くするようにテーブルの構造や記憶の割付けを行なうことがより簡単になるであろう．どこで誤りが起ろうとも，回復手順は可能なところはできるだけ自動的に行ない，オペレータやユーザの介入は最小限にしなければならない．ファイル全体を再ロードすることは絶対に必要な時以外は避けるべきである．あるファイル・ディレクトリを再ロードする場合

は，それが示しているファイル全体を再ロードしなければならない．なぜならば，古いポインタはダンプ後空白になり再割当てができる領域を示している可能性があるからである．インクリメンタル・ダンプ中に故障が生じた場合は，望ましいことではないが現在使用中のインクリメンタル・ダンプ・テープが停止させられることがある．回復手順はこのような状況も取扱う必要がある．

6.1.8 補助記憶機器の階層の取扱い

物理的にも経済的に考えても無限容量の記憶領域を単一機器上に実現することは不可能なので，この基本的な問題が生じてくる．したがって，アクセス速度や記憶容量の異なったいくつかのレベルのオンライン機器が必要となる．このオンライン機器の後にはディスク・パックまたは磁気テープのような取り換え可能な記憶媒体の記憶装置が補足されることがある．このような階層を図6.8に三角状のレベルとして表わしている．この図において，低速で安価な大容量の機器ほど三角形の底辺に位置し，高速で高価で容量の小さな機器ほど頂点に位置している．動作速度が適切でユーザが望むアクセス性能を発揮するレベルに情報が存在するようなアルゴリズムを考えることが最終の目標である．

まずディスクとテープだけのシステムを考えてみる．ユーザの明白な形での要求やシステム制御の下で，ファイルはディスクからテープに移される．どのファイルをテープに移すかを決めるためにシステムが利用できる情報には，ファイルがアクセスされた最新の時刻，与えられた期間でのアクセス頻度，あるいは割当てられた優先順位のような統計量がある．ディスクがある容量に達するとシステム・ルーチンは作業を開始し，テープに移せるファイルのファイル

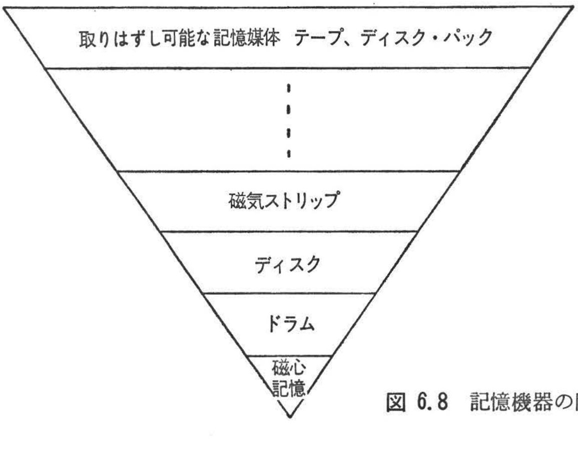


図 6.8 記憶機器の階層

・ディレクトリを探す．テープに移されたファイルはユーザが次回にアクセス要求を出した時にそのファイルを自動的に読出すことができるように適当な情報をディレクトリのエントリーに残しておく．ファイル・システムはユーザが明白な形でファイルを削除するまですべてのファイルを記憶しておかなければならない[13]．オンライン記憶装置の容量は限られているので，通常，各ユーザは記憶装置の固定された容量によって制約を受ける．

これらの考え方を機器の階層へ一般化する場合は，同じ概念が適用できる．ファイルがどの機器に入っているかを示すために，使用状況を基にしたアルゴリズムを開発する必要がある．各機器の**占有率**（occupancy factor）（一杯に詰まっている容量の割合）は，そのようなシステムを設計するにあたって重要なパラメータであり，シミュレーションまたはシステムの実験や測定を行なって研究しなければならない[12]．ファイルが参照されるとそのアクセス率カウントが1増されるが，時間がたつにしたがってこのカウントは減っていく．あるファイルのアクセス率がその機器にセットした値を下回る場合は，そのファイルは除去される可能性がでてくる．ファイルが直ちに除去されるかどうかはその機器および下位レベル機器の容量，またはそのファイルを下位レベルのいかなる機器にも格納したくないというユーザの拒絶指示などに依存する．拒絶特権に対しては，一般に高額の記憶使用料がそのような場合を制限する手段として課せられる．

機器が記憶容量の限界，すなわちソフトウェアのパラメータとしてセットされた常駐基準を越えた場合は，そこにあるファイルは利用率によって整理され，その一部が下位レベルの機器に移される．Multics では実現されれば，通常のページおよびセグメントのアクセス過程の一部としてその措置が動的に行なわれるはずである．このようにしてより低いレベルの機器に移されるような値の利用率を持つセグメントは，各ページが主記憶から補助記憶に戻されるようにページ単位に移される．

階層システムの目標は，最も頻繁にアクセスされる情報をより高速の機器上に保つことである．高速のアクセスを要求されるファイルがまれにしかアクセスされない場合に問題が生じる．このような状況を扱うためには，動的割付けアルゴリズムを拒否できる管理手順と注意深い課金方法を設定する必要がある．通常は比較的低速の機器に常駐するが，ユーザがシステムを使う間は一時的に高速機器に移される情報もある．

本節で基本ファイル・システムの概念についての概要を述べたので，これからは実際のI/O動作を含むより詳細な概念を調べることにする．

6.2 一般の I/O

6.2.1 概　　要

I/O システムは広範囲の機器によく対応しなければならず，詳細設計の段階は極めて複雑である．この複雑性は色々な形式の機器とのコミュニケーションを取扱う上での多くの特別な状況によって生じる．できるだけ概念的に単純な設計を行なうために，設計者は機器に依存したなるべく多くの特性を別々のルーチン（しばしば**機器駆動ルーチン**と呼ばれる）に分け，機器に依存しないより一般的なルーチンでこれらのルーチンを結合すべきである[11]．この問題に対する一解決法は，一般的なルーチンに使われるように設計されたテーブル内にできるだけ機器に依存した特性を格納することである．

I/O 機器を取扱う際に以下に示す4つの主要な機能があることがわかる：

1. I/O 機器，補助記憶，および主記憶間で転送される情報のバッファリング．
2. 割込み，すなわちプロセッサ信号に対する他の機器の取扱い，およびそれらとシステムの残りの部分とのインタフェースの取扱い．
3. I/O リソースの確保と割付け．
4. あるユーザあるいはプロセス専用のリソースに対する他のユーザあるいはプロセスの干渉からの保護．

以下の節でこれらの機能を実現する上で有用な概念や機構について述べる．これから述べる多くの機構は多くのシステムで種々の形式をとっているが，一般的なものである．特にL. P. Deutsch，B. W. Lampson 等によってプログラムされ XDS-940 に載せられているものは最もよく知られている．

6.2.2 バッファリング

I/O に関する概念で重要なものは**バッファリング**についてである．バッファは CPU と I/O 機器間の速度のつり合いをとるために使われる記憶領域である．プロセスがファイルをアクセスする間，システムはプロセスの要求を予期して，プロセスがファイルの情報を要求する時には必要な情報が主記憶内に存在するようにバッファを情報で満たすようにしておく．このようにして，プロセスは直ちに必要な情報をアクセスでき実行を続けることができる．バッファ

リングしないとそのプロセスはブロックされ，必要な情報が主記憶内に持込まれるまでシステムは他のプロセスに切換えなければならない．出力の場合は，プロセスがバッファを満たし，システムはバッファが一杯になった時に，自動的に必要な機器にバッファの内容を出力する．バッファが詰まっていれば，プロセスはバッファが空になるまでブロックされる．

システムは2種類のレコードを取扱う．すなわち，6.1.3節で述べた，(1) 論理レコードと (2) 物理レコードである．システムが維持するバッファの大きさは一般に物理レコードの大きさと同じで，それは機器の種類により異なる．たとえば，先に述べたように，ディスク上の物理レコードは，通常，セクタの大きさあるいはセクタをいくつか合わせた大きさである．またライン・プリンタにおける物理レコードはテキストの行に等しい．

しばしば2重あるいは多重バッファリングを用いて，システムが一方のバッファを空にするか満たしている間にユーザ・プロセスは他方のバッファを使って仕事をする．あるプロセスが現在使用中のバッファが空である場合，システムはポインタを切換えてシステムがその空のバッファを満たす一方，プロセスが第2のバッファをアクセスできるようにする．出力の場合，プロセスはバッファを満たし，そしてバッファは切換えられてシステムが一杯になったバッファを出力している間にプロセスは次のバッファを満たすようにする．2重，多重バッファリング技術は逐次ファイルのアクセスには便利であるが，ランダムにアクセスするファイルに対してはほとんどあるいは全く利益はない．図 6.9

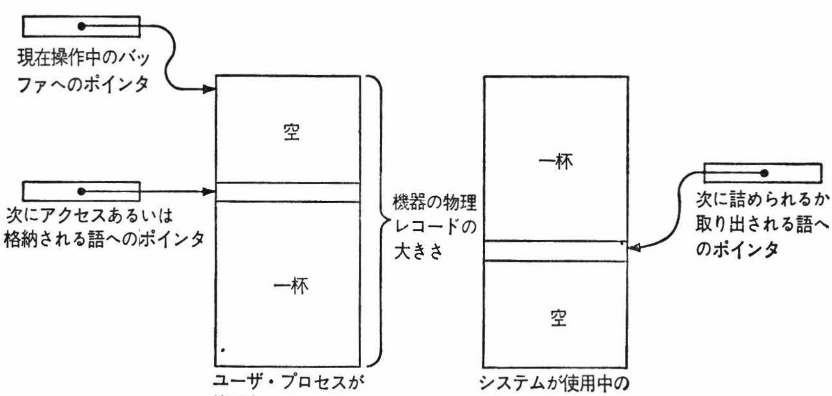


図 6.9　2重バッファ

に2重バッファの概念を示す．

これらのバッファ領域がどこに保持されるかという問に答えなければならない．システム内の各プロセスあるいはジョブは，一般にそれに関連した記憶領域を持っている．これはジョブあるいはプロセスの一部と見なせるが，ジョブやプロセスが直接アクセスすることはできない．この領域はプロセスと一緒にスワップされ，システムが一時記憶，定数の格納，テーブル，およびプロセスあるいはジョブをサポートするその他の機能を遂行するために用いる．たとえば，XDS-940 システムではこの機能のために各ジョブに対して主記憶の1ページが付随されることを5.2.5節で述べた．システムがバッファを維持するのはこの主記憶領域内である．この空間は限られているので，一度にバッファを通して行なえる転送はほんのわずかである．XDS-940 システムでは1つのジョブは3個の単一バッファを持つ．以下に述べるデータ転送開始に関連した機能の1つは，これらのバッファの1つを確保することである．

2重または多重バッファリング，あるいは標準の物理レコードよりも大きなバッファを必要とするようなデータ転送に対しては，プロセスは自分自身のアドレス空間中にバッファを定め，バッファ・アドレスをシステムに知らせなければならない．XDS-940 システムでは，標準の物理レコードより大きな情報ブロックをディスクに転送することができる．そのような転送をプロセスが要求する場合，プロセスは先頭の記憶アドレス，転送ブロックの長さ，ファイル名，およびファイル内の開始アドレスをシステムに与える．

プロセスがバッファに入っていない I/O 機器からの1語のデータを要求する場合，そのプロセスはブロックされるが，そのバッファを含む領域あるいはページはバッファが一杯になるまで記憶装置内に固定される（他のプロセスに割当てられない）．バッファが一杯になった時点でそのプロセスがプロセッサの制御を再び得るまでバッファ領域はドラム上に格納されるか，あるいは主記憶内に保持される．同様にプロセスが一杯になっているバッファへ出力しようとした場合も，バッファが空になるまでそのプロセスはブロックされる．プロセスのバッファ以外の部分がスワップされたとしても，バッファがプロセスのアクセスできる記憶領域内にある場合でも同様な処置がとられる．

バッファの取扱いに関連して数多くの管理上のタスクがある．たとえば，読み出しの際のバッファからプロセスへ，あるいは書き込みの際のプロセスからバッファに次に転送すべき語を指すポインタの操作，いつバッファが空になり一杯になるかのチェック，次に転送するファイルの物理ブロックの記録，およびバッファの大きさと記憶場所に関する記録などである．

6.2.3　割込み処理ルーチン

割込み処理ルーチンの設計　割込みあるいはそれと機能的に等価な機構が本書のいたるところで出てくるが，それはシステム設計においてそれらが中心的役割を持つからである．本節では5.2.4節で概説した多くの概念を復習し発展させる．議論は割込みに関してであるが，この概念はその他のコミュニケーション機構にも適用できる．割込みには3つの基本的な形式がある．すなわち，(1) 現在割当てられていたタスクの終了を示すもの，(2) サービス要求，(3) 誤りあるいは故障状態の指示である．3番目の形式の割込みは機器の種類によって独特な取扱いが必要なのでここでは議論の対象外とする．しかしながら，種々の誤りに対してはシステムに誤動作の報告する前に何回か動作を再試行する方法がとられる．この例としてデータ転送中のパリティ・エラーがある．そのようなパリティ・エラーが発生すると，同じ動作をくり返し行なう．この再試行は多くの誤りが一時的な原因で生じるという経験に基づいている．

最初の2つの形式の割込みは，スケジューリング・プロセスに対する重要な入力となり，一般にあるプロセスを呼び起こすべきことを示す．データ・チャネルや数種の機器が接続されたI/Oプロセッサが発生するような割込みは実行中のサービス機能に特有の意味を持つ．これらの機器で起こされた割込みはそのサービスを要求したルーチンによってのみ適切に解釈できる．それゆえ，その割込みを途中で捕える一般ルーチンを作り，チャネルあるいは割込みを発生している機器から得られる状態情報をすべて検索し，この情報をサービスを開始した特定の機器駆動ルーチン，あるいはサービスを要求している特定の機器を取扱うように設計された機器駆動ルーチンに送るように設計するのが一般的な方法となる．この機能を行なうためには，チャネル駆動ルーチンが既に開始しているサービスを何らかの方法で記録していなければならないことを意味している．またチャネル駆動ルーチンは特定の機器から割込みを受取った時，サービスを要求しているルーチンの中に転送するアドレスを格納しておかなければならない．このような機器個有の割込みルーチンは状態情報を解釈でき，誤りの条件があれば回復を試み，一連の動作が必要な場合は次の動作を開始する．たとえば，ディスクではアドレス・シーク要求の後にデータ転送が続く．機器固有のルーチンはサービスを要求したプロセスあるいはサービスを要求しているプロセスとコミュニケーションするために使われるコミュニケーション・セルの記憶場所を記憶しておかなければならない．割込み処理ルーチンは動作終了またはその他の条件をサービスを要求しているプロセスに知らせるため

に復帰情報を記憶しておかなければならない.

I/O の詳細例　一例を挙げれば，これらの考え方をいくつかの概念を付随して説明することができよう．逐次ファイルFから入力を行なうプロセスPを考える．チャネルを通してシステムに接続されているディスク装置上にそのファイルがある．チャネルは1本の割込み線を持っている．システムは1つの単一バッファを備えている．図 6.10 に考えられるコミュニケーション経路を示している．バッファが空でプロセスPが入力要求を出す場合，Bと呼ばれるバッファの処理を行なうルーチンがプロセスPをブロックする．そこでルーチンBはファイル・システムからの入力を要求する．この入力要求はディスク・サービス要求待ち行列のエントリーに置かれる．ルーチンBは現在サービス中のレコードを保持している．その待ち行列のエントリーにはバッファの記憶場所とファイルFの次の物理ブロックに対するディスク上の記憶場所が格納されている．後者の情報はルーチンBが適切なテーブル（次節で述べる）から得たも

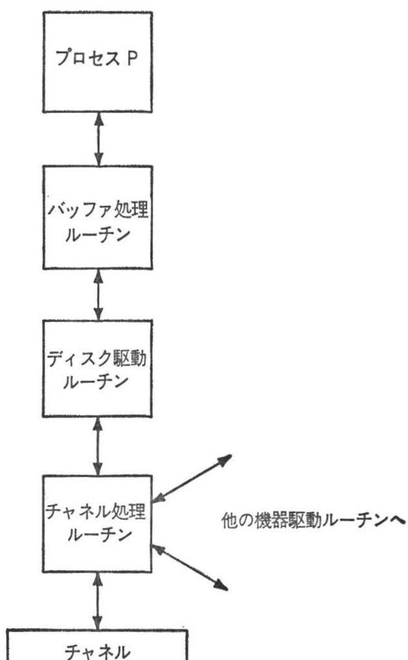


図 6.10 I/O 動作に対するコミュニケーション経路

のである．ディスクを駆動するのに使うルーチンはディスク待ち行列からエントリーを取り出し，チャネル処理ルーチンに適切な I/O コマンドを出す．

ファイルFのアクセス要求が待ち行列の先頭に到達すると，ディスク駆動ルーチンは最初にチャネル処理ルーチンに対して必要なセクタに読み取りセンサを位置付けるためのシーク・コマンドを出す，ディスクから割込みがかかると，ディスク駆動ルーチンはチャネル処理ルーチンに制御を特定の記憶場所に戻すように知らせる．シーク動作が終了すると割込みが発生する．チャネル処理ルーチンはチャネルからの状態情報によって割込み処理を行なう．状態情報を調べると，その割込みがディスク制御装置から来ていることが分かる．以前に送られた情報を用いて，チャネル割込みルーチンは制御をディスク駆動ルーチンへ戻す．

そこでディスク駆動ルーチンはファイルFからプロセスP用のバッファに物理ブロックを読み込む I/O コマンドを出す．ディスク駆動ルーチンはまたチャネルに次のブロック転送終了割込みがかかる時に制御を返すように知らせる．物理ブロックが転送し終ると，チャネル語計数レジスタは零になり，チャネルはブロック転送終了割込みをかける．チャネル割込み処理ルーチンはチャネルから状態情報を受取って，割込みのサービスを行なう．割込みがブロック転送終了信号であることが判断されると，制御はディスク駆動ルーチンに戻される．ディスク駆動ルーチンは制御を取り戻すと，どのルーチンの処理を行なっているかを調べ，この例ではルーチンBであることを知る．制御は転送が正常に行なわれたことを示す情報と一緒にルーチンBに戻される．

ルーチンBはプロセスPに知らせることができるように転送誤りが生じたかどうかを知らされなければならない．誤りの指示はチャネル割込み処理ルーチンからディスク駆動ルーチンへと移っていたはずである．ディスク駆動ルーチンBに誤りを知らせる前に I/O 要求を数回試みる．

正常な転送が行なわれると，ルーチンBはプロセスPを呼び起こし，プロセスPは再びファイルFからの情報を要求する．誤まった転送が行なわれた場合，プロセスPは呼び起こされはするが，誤り処理の点から実行を始める．すなわち，ルーチンBは全く異なったプロセスを呼び起こし，誤り状態をユーザに知らせるメッセージを印刷するか，あるいは何か別の操作を行なう．この例は種々のレベルの I/O と割込み処理ルーチンとの間で必要なコミュニケーション機構の形式を示している．

割込みの問題に関連して，割込みによって終了あるいは条件を知らされない場合に対する状態検査をいかに取扱うかという問題がある．XDS-940 を一例

にとると，テープの巻戻し要求の場合がある．テープが既に巻き戻されていると，割込みは発生しない．この条件はオペランド・コードに示される I/O 条件が満たされる場合にスキップする特別な形式のテスト命令で検査することができる．オペレーティング・システムがこの条件を絶えず検査するようにしてしまうのは望ましくない．このために使われる技術は，巻き戻しを開始させるテープ・ルーチンがシステム・プロセスを生成するシステム・ルーチンを呼び出すようにする．そのシステム・プロセスには巻き戻し終了条件を調べる仕事を割当てる．このシステム・ルーチンによって呼び起こされるまでテープ処理プロセスの通常の待ち行列に置かれ，クロック処理ルーチンにより2，3秒毎に呼び起こされ，テープ巻き戻し条件を調べる．この手法は Multics および XDS-940 システムで用いられている．

まとめ　割込み処理プログラムは特定の機器はもとより，スケジューリング機能にもよく適合するように統一的に設計されなければならない．プログラムを統一化するためには，すべての機器に対するサービス要求を標準的な形にし，復帰のコミュニケーションができるように呼出しルーチンに適当な識別子を付けてやる必要がある．

6.2.4 I/O 機器の割付け

専用機器と共用可能な機器　機器には2種類の形式がある．すなわち，(1) 専用機器，および (2) 共用可能な機器である．**専用機器**はたとえユーザが連続的に使用できなくとも，与えられた時間の間に1人のユーザに最も効率良く割付けられる．テープ駆動装置，ラインプリンタおよびその他のプリンタ機器，カード機器，および光学的走査機器などがこの部類に入る．**共用可能な機器**は一時に1つのプロセスのみにアクセスを許す一方，個々のプロセスに対してサービスを高速に行なえ，他のプロセスのサービス要求に対して敏速に切換えられる．ドラム，ディスク，およびデータ・セルのようなオンライン補助記憶装置がこの部類に入る．

専用機器は一定時間および期間にわたってそれらを特別に確保するための予約が必要である．専用機器の予約は計算機によらない管理手続きによって取扱うことができる．またユーザはログ・インし，システムにコマンドを与え，所望の機器が空いた時を見つけ出し，システムに予約をとり，ログ・アウトすれば良い．スケジュールされた時間にユーザはログ・インし，たとえばユーザが予約したテープ駆動装置をユーザ・プロセスで利用したいということを要求す

る．システムは予約スケジュールを調べ，リソースを割付ける．Multics システムではこの方式をとることができる．

XDS-940 システムでは，システムは機器要求を先着順サービスで処理するので，予約は計算機システム外で行なわなければならない．たとえば，あるユーザがファイルをプリンタへ転送しようとし，そのプリンタが使用中であった場合は．“プリンタ使用中”を示すメッセージがユーザの端末に返送される．そのユーザは要求を改めて後で出さなければならない．プリンタに対するサービス要求待ち行列を設計し，プリンタが使用中の場合はユーザにこのことを知らせ，その要求が受入れられた時にその情報がユーザに送られる．

共用可能な機器はその共用がユーザにわからない割合で異なったプロセスに割付けられる．いかなる特別な予約も必要ではない．ここで専用および共用可能な機器の双方に応用できる数多くの概念について論じよう．

機器のオープンとクローズ　　機器の割付けは**機器またはファイルをオープン** (open) するプロセスを通して行なわれる．ある機器を使うプロセスが終了するとその機器は**クローズ** (close) される．これらの機能はユーザ・プロセスに見えないシステム・ルーチンで処理される．専用機器がプロセスにオープンされると，その機器がクローズされるまで他のプロセスはそれを使うことはできない．このことは共用可能な機器に対しては当てはまらない．システムはあるプロセスに対して専用あるいは共用可能な機器をオープンする場合，その機器に関する情報を含むテーブルを調べる．

この種のテーブルはしばしば**機器制御ブロック**あるいは**機器制御テーブル**と呼ばれ，その中にはその機器が接続されているチャネルの番号，機器番号，機器が字形式か語形式かの指示，バッファが必要かどうかの指示，必要の場合のバッファの大きさ，機器に接続されているサブユニット数，機器がランダムか逐次的か，機器駆動ルーチンのアドレスなどの情報を含んでいる．

機器が現在使用中であれば，機器テーブルはこの事を示している．機器が使用中の場合，システム設計によって次の2通りの手続きがとられる．(1) プロセスは実行を止められ，準備リストに入れられる．準備リストで順番が回ってきた時にその要求を出す．あるいは，(2) 機器要求は待ち行列に入れられ，機器が割当てられるまでこのプロセスはブロックされる．機器が使用中であることを示すテーブル中の記録は，実行中の情報転送に対して他のいかなるプロセスも干渉できないことを保証する保護機構になっている．言い換えると，アクセスに対するすべての要求をたった1つのルーチンあるいはテーブルを通すこ

とによって保護が得られる．

機器，ファイル，プロセスなどの実体に論理的に関連するすべての情報は論理レコードあるいはエントリーと考えられる．レコードはリスト処理を用いて互いにつなぎ合わされるか，あるいは記憶装置の連続した領域に格納される．関連するレコードの組はテーブル，待ち行列，あるいはリストと呼ばれる．リスト処理を使った例を5.2.6節で示した．ここでの議論では，論理レコードあるいはエントリーは主記憶内の連続した領域にあるものと仮定する．主記憶中にそのようなレコードを構成するために2つの一般的な方法がある．1つは図6.11に示すように，レコードに対する領域を隣接した記憶場所に格納する方法である．一般にこのようなテーブルはポインタをインデックス・レジスタ中のレコードの第1語に置き，所望の領域を含む語のレコード内の相対記憶場所をアドレス・フィールド内に格納することによってアクセスされる．必要とする特定の情報領域は通常コーディングをする時にはわかっているが，必要なレコードは実行中に変わるのでこの方法が使われる．

第2の構成法は，図 6.12 に示すように，レコードに対する語をいくつかの1次元アレーにおける同一の相対記憶場所に置くようにする．与えられたアレー中のすべての語は同じ構造をしている．このテーブルは必要なレコードに対する相対ポインタをインデックス・レジスタに格納し，アドレス・フィールドに必要なアレーのアドレスを格納することによってアクセスされる．

専用機器は空くまでプロセスにはオープンされない．オープン要求は待ち行列に入れられる．共用可能な機器はたとえそれが使用中であってもオープンさ

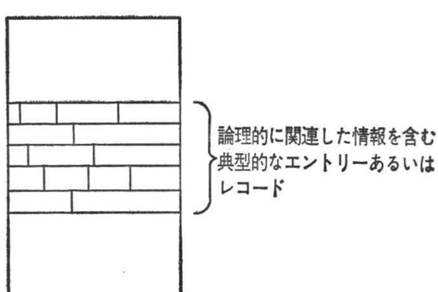


図 6.11 連続語で構成されたレコードの領域

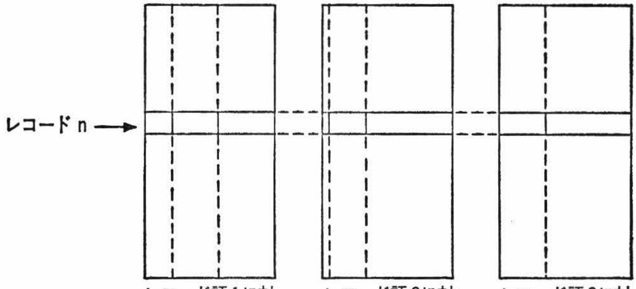


図 6.12　数個のアレーの語で構成されたレコードの領域

れ得るが，そのアクセス要求は待ち行列に入れられる．I/O 機器に対する待ち行列の取扱いは概念的には CPU に対する待ち行列の取扱いと同じである．I/O 機器に対する待ち行列は単純な先着順サービスか，あるいはサービスを要求しているプロセスの優先順位に基づいたものである．

ディスクのような共用可能な機器に対するアクセス要求の待ち行列は，プロセスの優先順位だけではなく，機器の特性をも考慮するために，より精巧な待ち行列スケジューリング・アルゴリズムを持っている[8]．たとえばディスクに対するアクセスのスケジューリングは，ディスクの全体的な利用が最適になるように現在の読み書きセンサの位置を考慮する．

待ち行列のエントリーに必要な情報は，要求されたサービスを適当に指示するための情報，サービス要求をしているプロセスとその優先順位ならびに呼び起こし信号の種類，サービスの終了時に送るべき指示，などである．最後の機能は，5.2.4節で述べたように，非常に簡単なものである．サービスが終了した際に0より大きな値をセットすべき記憶場所のアドレスに送る程度である．

機器あるいはファイルをオープンするプロセスは，ファイルを定義するという考えとしばしば混同されている．ファイル・ディレクトリ中にエントリーが作られる時にファイルは定義される．明白な形で削除されるまでファイルは定義されたままである．**オープンする**という概念は特定のデータ転送あるいは転送シークエンスに関連しており，バッファ領域のように転送のために必要なリソースの割当ておよび必要な機器へのアクセスを必要とする．6.1.1節で述べたように，オンライン補助記憶以外の機器は特定の名前を持ったファイルとして取扱えるので，機器およびファイルという用語はしばしば交換されて使われる．端末を使っている間，ユーザ・プロセスは数回にわたって特定のファイル

をオープンしたりクローズしたりする．

必要な機器が未使用かあるいは共用可能な機器である時，その機器は要求を出しているプロセスに割当てられ，そのプロセスに対してオープンされる．必要なバッファの大きさを決定するために，その機器に関する機器制御テーブルがその他の情報と同じように調べられる．またファイル・ディレクトはファイル・インデックス・ブロックとファイルに関する情報の記憶場所を決定するために調べられる．この特定の転送に関連して**ファイル制御ブロック**がセットされ，その中には実際に要求された I/O を行うために一般の I/O および割込み処理ルーチンに必要なすべての機器，ファイル，プロセスに関係した情報が含まれている．

数語からなるこのブロックの中の情報は，転送に使われるバッファのアドレス，機器がディスクあるいはその他のオンライン補助記憶の場合はインデックス・ブロックのアドレス，ファイルが読み出し専用が逐次あるいはランダムかの指示，実際に割当てられた機器の機器番号，転送方向が読み出しか書き込みかの指示，転送状態情報，チャネル番号，などである．プロセスが明白な形で要求を出し，またはバッファの操作をするシステム・プロセスがこの機器あるいはファイルに対してデータ転送を行なう度に，ファイル制御ブロック内の情報がI/O要求を設定するために使われる．

ユーザは記号名称によってファイルを参照する．システムは番号でファイルを参照する．通常，このファイル番号は単にファイル制御ブロックのテーブルに対するインデックスである．特定のファイルのファイル番号はファイルのオープンからクローズの間で変わる．記号名称はユーザが変えるまで固定されたままである．同時にオープンできるファイル数は必要な機器，使用可能なバッファ数，使用可能なファイル制御ブロック数に依存する．機器の数はハードウェア構成によって一定である．バッファ数はバッファ空間の割当て方に依存した設計上のパラメータであり，ファイル制御ブロック数は可変パラメータである．一連のアクセス処理が終了するとその機器あるいはファイルは**クローズ**される．機器をクローズすることは割付けられていたリソースを解放することである．

まとめ　以下に重要な概念である専用機器および共用可能な機器，機器あるいはファイルのオープンとクローズ，機器に関係した情報テーブル，ファイル制御ブロック，そしてファイル番号について概説した．これらの概念は機器に独立な統一された I/O システムを設計する際の中核となるものである．こ

れらの概念の多くは遠隔機器を取扱う際にも適用できる.

6.2.5　端末の I/O

概　要　ここで述べる基本的な概念は多くの形式の端末にも適用できるが，本書では重要と思われる**字形式**（character-oriented）遠隔端末のみについて考える．汎用グラフィック端末あるいは小型遠隔計算機に接続される端末などが特殊な研究分野である[7]. 端末のコミュニケーションを扱うのに2つの主な方法がある．すなわち，(1)単一字を伝送する方法，および(2)一時に1ブロックの字を伝送する方法である．前者は後者よりも会話形システムを設計する上で融通性があるように思えるが，字を取扱うためのシステム時間が多く必要である. XDS-940 システムはテレタイプ I/O に対して単一字方法を使っている. 種々の字形式グラフィック端にはブロック転送が必要である． IBM の Quiktran システムは一度に1行の単位で伝送する．各方法を取扱うためのソフトウエアには共通点が多いが，ここでは単一字を伝送する方法のみについて論じる.

XDS-940 では，コミュニケーション・インタフェースは CPU の直接 I/O 線に接続され，字毎の I/O には約 300 マイクロ秒の処理時間が必要である. 一般に使われる別の方法は GE-645 でも使われているが，マルチプレクサ・チャネルを介して直接主記憶に字を入力する方法である．通常16から32台のテレタイプが接続される XDS-940 のようなシステムでは端末の I/O を扱うためのオーバーヘッドはさほど大きくはない．しかし50から 200 台の端末を使った大規模なシステムにとってはこの方法はあまりにも非能率的である．大規模システムの設計者達は端末のコミュニケーションを取扱う個別の小型プロセッサを使うようになってきた．これらのコミュニケーション・プロセッサは自分自身の記憶装置を持ち，またシステムの主記憶をアクセスする．機械語から字をパックしたりアンパックする効率的な命令もシステムの字処理に費やす労力を軽減する上で重要になる.

種々の字形式端末および機器を関連したデータ速度で使えるようにするには，設計に際して端末インタフェースのハードウェアならびにソフトウェアをモジュール化することが必要である．種々の字セットを持った端末が利用できるが，それらはソフトウェア・ルーチンが認識しなければならない．各機器はその操作に必要な特別の制御コードを持っている．利用できる機器のデータ速度は 10 ないし 240 字の範囲にわたる． 主要な端末としてテレタイプを使った XDS-940 のようなシステムでは，その他の端末を付け加えるためにはソフト

ウェア・ルーチンを，そして字形式グラフィック端末の速いデータ速度に合わすためにハードウェア・インタフェースを大幅に手直ししなければならなかった．設計目標はたとえあるにしてもほんのわずかなシステム変更で種々の端末が使えるようにシステムをモジュール化することである．

バッファリング　端末の I/O と同様に一般の I/O に関して重要な概念は，ユーザ・プロセスは I/O 機器と直接コミュニケーションするのはまれで，バッファとコミュニケーションするのが普通であるということである[10]．端末の I/O の場合，ユーザ・プロセスが字を端末に送る命令を出すと，システムはこの命令を翻訳して実行し(実際にはシステム呼出しが作られる)，端末に送る字をこのプロセスに関する出力バッファの中に入れる．その後，すべての端末の I/O 動作を管理しているシステム・ルーチンがバッファからその字を取出して端末に送る．あるプロセスが端末へ多量の出力を出す場合，バッファがあふれてしまう．システムは出力バッファの詰まり具合を監視し，あふれそうになるとそのプロセスをバッファが空になるまでブロックする．

入力の場合，端末からの字はシステムによって自動的に入力バッファにロードされる．プロセスが実行中に字入力命令を出すと，システムはこの命令を翻訳して実行する．システムは字を入力バッファから取出し，約束によって割当てられたレジスタあるいはユーザが指定した記憶場所に格納する．バッファに字が入ってない場合は，十分な数の字がバッファに到着するまでプロセスはブロックされる．

字はユーザ・プロセスには非同期的に計算機に入って来たり出て行くので，これらのバッファは主記憶内に常駐しなければならない．高速転送機器は低速機器よりも大きなバッファを必要とする．バッファ割当ての一方法は各形式の機器に前もってバッファを割当てておくことである．バッファの大きさや数に関係してバッファは主記憶空間をひどく浪費することになるかもしれない．この問題を軽減するために，未知あるいは可変長の領域が必要となる同様な問題に直面している他の応用分野からリスト処理技術を借りてくる方がよい．この方法を使えば，ある長さ，たとえば10から20語のブロック群に細分された全バッファ領域を前もって割当てることができる．動作中の各端末には必要に応じて最大語数のブロックを1つ割付ける．その時点で出力か入力によってプロセスはブロックされるかあるいは呼び起こされる．リスト処理を使えば全記憶容量は小さくなり，しかも個々の端末に必要な可変長のバッファを実現できる．

端末のバッファは通常リング構造になっている．たとえば，入力バッファで

あるポインタＩは次の入力要求時にどの字をプロセスに送り出すかを示し，他のポインタＲは次に端末から受け取る字を格納すべき場所を示す．ポインタＲとポインタＩが同一場所を示す時はバッファは一杯になっている．このような状態あるいはそのことを示す警告をチェックする必要がある．この概念を図6.13に示す．同じような構造が出力バッファにも使われる．

端末番号　各端末には１つの端末番号が付けられる．ジョブ番号やファイル番号と同じように，端末番号は端末の I/O に関連するテーブルを引いたり適当なバッファをアクセスするためにシステムが使う番号である．端末がシステムと接触する際にこの番号が割当てられる．各端末は制御のために使われる種々の状態情報を含むテーブルを持つ．すなわち，端末が他の端末にリンクしているか否かの指示（以下に述べる），他の端末がこの端末にリンクしているか否かおよびリンクしている端末の端末番号，データ・セットの状態，ブレーク・キャラクタ・セット，必要なコード変換形式，およびその他の状態情報が含まれる．

システム呼出し　端末の I/O に関連したもう１つの重要な概念は，どのようなプロセスが実行されていてもシステムあるいは次に高いレベルのプロセスに戻るための機構である[10]．ユーザが実行中のプロセスから抜け出したいという場合は，プロセスがループ内で行き詰まったり，不必要な出力を長いテーブルに打出し始めたり，あるいは制御している問題解決の一段階が終った時である．

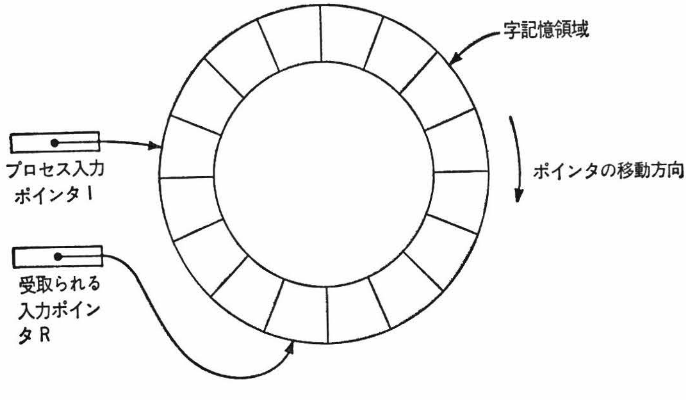


図 6.13　論理的にリングとして構成された入力字バッファ

XDS-940 システムではこの機能を字コードを使って行なう．入力字ルーチンがこの特別な字を調べる．ソフトウェア割込みをセットすることにより，制御はシステムあるいは特定のプロセスに渡される．この特別な字を何度も打込むかあるいは十分接近して続ければ制御はシステムのルート・プロセスに戻る．その他のシステムではこの機能のために特別なスイッチあるいはボタンを使用している．

通信回線の故障　一時的な通信回線の故障中に起こり得る種々の場合を取扱うために入念な端末プログラミングが必要である．回線を切離す必要が出た場合にシステムがこの状況を検出できることが重要である．システムがそれを検出できないと，ユーザは依然としてログ・インの状態にあり，リソースを使い続ける．その時に使用中のファイルもまた危険にさらされる．システムは自動的にユーザをログ・アウトする前に，その故障が一時的なものではない事を確認するためにしばらく待機しなければならない．

ブレーク・キャラクタ　ユーザのプロセスが1字のコマンドを受取れるようになっている場合，プロセスが呼び起こされる前にバッファ内にあるべき字数に対して制限を設定してもうまくいかない．XDS-940 および他のシステムではこの問題を避けるために**ブレーク・キャラクタ**（break character）という概念を使っている．ある種の字，または場合によってはどのような字でも良いが（ユーザがシステムに与えることのできるコマンドによってブレーク・キャラクタ・セットの数や範囲が設定される），その字が受取られるとユーザ・プロセスが呼び起こされ，バッファから字が取り出されるようになっている．

呼び戻し機能　端末の I/O に関連したもう1つの重要な概念に**呼び戻し機能**（echoing）と呼ばれるものがある．940 システムにもその他のシステムと同じように呼び戻し機能がある．この機構は端末の送信ならびに受信装置は物理的には同じフレーム内にありながら論理的には互いに異なっているという事実を利用している．呼び戻し機能においてはユーザがキーを押すとその字コードが中央の計算機に行き，ユーザの操作では端末にその字は印刷されない．システムはその字を受け取り，他の出力とは異なるバッファに格納する（プロセスが生成する出力と混同しないように）．その後システムは呼び戻しバッファを出力する．図 6.14 に呼び戻し機能の概念を示す．テレタイプはキーを押した時に印字されるように入力と出力の機構を直接結合するスイッチを備えて

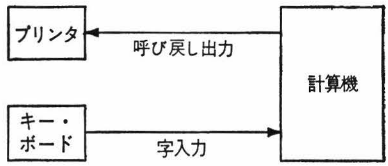


図 6.14　呼び戻しコミュニケーション

いるのに，なぜより高価な呼び戻し手順を行なうのであろうか．呼び戻し機能には2つの利点がある．すなわち，(1) 呼び戻し処理は誤り検出能力を与える．正しい字が戻ってくればユーザは入力した字が正しく受取られた事を知ることができる．データ入力に対するこの種の保証の有用性は明らかである．(2) 呼び戻し処理はコード変換あるいは翻訳機能を与える．ユーザが1字を送り，その字と同じものを受取るか，異なった字を受取るか，あるいは全く送り返されないような機能である．

3.4.3 節で述べたような蓄積交換技術を使うコミュニケーション・システムでは，1字の送出と呼び戻しの遅延時間が呼び戻し機能を混乱させてしまうことがある．なぜならば入力時に前の呼び戻し出力が帰ってくることがあるからである．その場合は半2重モードの通信回線にキーボードとプリンタ機構を直接結合したものがよい．システムは呼び戻し機能を持つ半2重および全2重の両方のモードを受け入れるように設計されていると便利である．これらのモードはシステムへのコマンドで切換え可能である．

リンク機能　大低のシステムにあるもう1つの特徴は端末を入出力時に互いに**リンク**できる機能である[10]．入力リンク機能はソフトウェアで端末を互いに結合し，1台の端末でキーを押せばその端末ならびにリンクしている端末に同じ字が出るようになっているものである．同様に端末に対する出力リンク機能はプロセスの出力をそれ自身の端末およびリンクしている端末に出す．このような機能には色々な利用法がある．たとえば授業をする場合や2人以上である問題を共同で解く場合に利用できる．

単純な編集機能　テレタイプ・ルーチンが単純な編集の雑用を処理し，種々のサブシステムが別個に編集ルーチンを持つ必要性をなくせば，システムの応答は著しく高速化できる．ユーザが誤ったキーを押した時はその字あるいは1行全体を削除するための編集用の字を送ることができる．この頻繁に使われ

る編集の仕事を端末入力ルーチンが行なうことができれば，サブシステムについてスワッピングをせずに直ちに編集作業ができる．サービス要求の大部分はこの種の単純な編集作業であり，数個の命令でできる．この種の単純な編集機能がテキスト・エディタ，会話形コンパイラ，応用ルーチンの中で行なわれれば，これらの単純な要求を満足するためにシステムは主記憶とのやりとりですぐに手一杯になってしまう可能性がある．通信回線を接続するために小型計算機を使えば，ファイル操作のみならずより複雑な編集機能を CPU に干渉せずに実行できる．また通信用小型計算機を使うことにより，広範囲の端末と転送速度を容易に取扱えるようになる．

まとめ　システム設計者は端末コミュニケーション・システムを設計する際には特別の注意を払う必要がある．融通性に富んだ強力なシステムはユーザの応用設計に大きな手助けとなり，新型の端末機が市販された時にそれを容易に追加できる．システム機能に関連するものとして，バッファリングの概念，単純な編集機能，および通信回線の誤動作を取扱うための提案などを述べた．ユーザに有用な技術は，自分のプロセスが何を行なっていてもシステムを呼び出す能力，ブレーク・キャラクタの利用，他の端末とリンクする能力，および半2重あるいは全2重でコミュニケーションできる能力（呼び戻し機能）などである．端末の I/O を取扱うために小型の通信用計算機を使えば多くの有利性が出てくるように思われる．

6.3 ユーザ・インタフェース

ある端末のユーザとシステム間のインタフェースをサービスするルーチンの多くは，通常主記憶内に常駐していない．ユーザ・インタフェースとその他の非常駐ルーチンは XDS-940 システムでは**管理ルーチン**（executive）と呼ばれるリエントラント・プログラムに一まとめにされている．Multics システムではいくつかのセグメントから成り，あるものは主記憶に固定された**常駐セグメント**である．その他は呼び出すと得られる**非常駐セグメント**である．ユーザと接する特定のセグメントは**コマンド・システム**と呼ばれる．

次のようなタイムシェアリング用のコマンドがある：

1. システムへログ・インおよびログ・アウトするコマンド
2. ファイルの作成，削除，操作をするコマンド

3. サブシステムにアクセスするコマンド（サブシステムもそれ自身のコマンド言語を持てる）
4. システムの状態を判断するコマンド

一般にオペレータは特権コマンドが使える：

5. 新しいユーザの定義，ユーザの削除，ユーザの特権の変更，課金情報のアクセス，および一般のユーザが得ることのできるシステム状態情報よりも詳しいものをアクセスするコマンド

ユーザがシステムにログ・インすると，そのユーザのために種々のテーブルが作られ，ユーザはコマンド・システムを利用できるようになる．コマンド・システムはリエントラント・コードで書かれており，論理的には各ユーザは自分自身のコマンド・システムを所有しているように思う．ユーザはいつでもコマンド・システムにアクセスできる．仮にユーザがサブシステムまたは自分のプログラムの1つとコミュニケーション中であっても，システムがそのために便宜的に割当てた1字を押せば，コマンド・システムにアクセスできる．テレタイプ処理ルーチンがこの字を知るとコマンド・システムが主記憶中にない場合にはそれを主記憶に持ち込むルーチンに飛び越しが起こる．

コマンド・システムはコマンドを形成する一連の字だけを集め，そのコマンドを解読し，解読したコマンドを取扱うルーチンへ飛び越しを起す．コマンドに対する引数もすべてそのルーチンに渡す．コマンドを取扱うルーチンは直接コマンドを取扱うか，あるいはその機能を実行するプロセスを生成する．

たとえば，コマンドがサブシステムの呼出しである場合は，呼び出しているユーザの適当な記憶テーブルをセットして，そのサブシステムをユーザに割当てる．そのサブシステムが主記憶中にない場合は，適切な I/O ルーチンを呼出し主記憶に読み込む．サブシステムをセットするのに必要な情報は特別のテーブルに格納される．

コマンド・システムは当然のことながら特別なプログラミング言語，すなわちコマンド言語に対する解釈ルーチンと見なせる．このような見方をすると，Multics におけるようにコマンド言語プログラムの考え方が出てくる．ある種のタスクを遂行するために，ユーザは会話中に点々と一連のコマンドを必要とする場合が多々ある．一例は1つのシステムを生成するのに必要なファイルを主記憶にロードし，アセンブルし，ドラムにロードする場合である．コマンド言語プログラムを1つのファイルとして書いたり格納したりでき，しかもコマ

ンド言語解釈ルーチンがこのファイルの実行を指示できるようになっていれば，ずっと楽になる．この方法を使えば，コマンド言語を詳しく知らない人でも特殊目的のプログラムをアクセスするための簡単なプログラムを開発できる．コマンド言語はユーザがシステムと接触する重要な点の1つであるので，構文と機能の双方についてその設計には十分注意を払う必要がある．システムに対するユーザのインタフェースのすべての面が使いやすさおよび学びやすさの観点から注意深く考慮されなければならない[9].

参 考 文 献

1 **Daley, R. C., and P. G. Neumann:** A General Purpose File System for Secondary Storage, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, 1965.

2 **Organick, E. I.:** A Guide to Multics for Subsystem Writers, *Project MAC Doc.* March, 1967.

3 **Daley, R. C., and J. B. Dennis:** Virtual Memory Processes, and Sharing in MULTICS, *Commun. ACM*, vol. 11, no. 5, p. 306, May, 1968.

4 **Corbato, F. J., and J. H. Saltzer:** Some Considerations of Supervisor Program Design for Multiplexed Computer Systems, *IFIP Conf. Proc., Edinburgh*, August, 1968.

5 **Oppenheimer, G., and K. P. Clancy:** Considerations for Software Protection and Recovery from Hardware Failures in a Multiaccess, Multiprogramming, Single Processor System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 29–37, 1968.

6 **Madnick, S. E.:** A Modular Approach in File System Design, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 1–13, 1969.

7 **Sutherland, W. R., and J. W. Forgie:** Graphics in Time-sharing: a Summary of the TX-2 Experience, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 629–635, 1969.

8 **Denning, Peter J.:** Effects of Scheduling on File Memory Operations, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 9–21, 1967.

9 **Hargraves, R. F., Jr., and A. G. Stephenson:** Design Considerations for an Educational Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 657–664, 1969.

10 **Lampson, B. W., et al.:** A User Machine in a Time-sharing System, *Proc. IEEE*, December, 1966.

11 **Ossana, J. F., et al.:** "Communication and Input/Output Switching in a Multiplex Computing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 231–242, 1965.

12 **Ver Hoef, Edward W.:** Design of a Multi-level File Management System, *Proc. ACM Natl. Conf. 1966*, p. 75.

13 Wilkes, M. V.: "Time-sharing Computer Systems," American Elsevier Publishing Company, Inc., New York, 1968.

14 Hoffman, L. J.: Computers and Privacy: A Survey, *ACM Computing Surv* vol. 1, no. 2, pp. 85–103, June, 1969.

15 Dodd, G. G.: Elements of Data Management Systems, *ACM Computing Surv.* vol. 1, no. 2, pp. 117–133, June, 1969.

16 Lichtenberger, W. W., and M. W. Pirtle: A Facility for Experimentation in Man-Machine Interaction, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 589–598, 1965.

17 Smith, M. G., and W. A. Notz: Large-scale Integration from the User's Point of View, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 87–94, 1967.

18 Ha, E. P. L.: Unpublished notes on time-sharing file systems, Shell Development Co., 1968.

第 7 章

測定，信頼性および回復

7.1 システムの測定と評価

7.1.1 概　　要

設計過程では次々に種々の決定を行なわなければならない．タイムシェアリング・システムの設計ではどの機能をハードウェアで実現し，どれをソフトウェアで実現するかを決める必要がある．これらの決定を行なってから，ハードウェアとソフトウェアの目標をどのように満足させるかについてさらに色々決めなければならない．設計の初めから設計者は価格，一般性，速度やその他種々のつり合い問題に直面する．

設計が完成し具体化を進めている際に構成要素やモジュール，および最終システムを検査し評価しなくてはならない．一般に，ここで学んでいるような複雑なシステムにおいては，モジュールの重なり合った相互作用は設計段階で十分に予想できるものではない．それゆえ，システムが論理的に正しくても性能を改善する余地がおそらくあるだろう．各設備の使用環境はそれぞれ異なるので，システム構成およびリソース割付けアルゴリズムのパラメータを選択するという面でその設備に対する調整が行なわれる．設備における仕事の量が増大し多様化してきた時，再構成に関する判断が再び要求される．システムを構成することは難しい設計問題であり，一般に良く理解されていない．独創的な設計者は自分の努力の結果を分析したいと思うが，次のシステム作成段階でも結局いままでと同じ過程を繰り返す．

必要な判断と評価をいかにすべきであろうか．その答は設計の各段階において測定し評価するということである[1-3]．以下の意見はいま挙げた3つの文献で実証された経験に基づいている．設計の過程は理論的あるいは概念的な要素および経験的な要素から成り，それらの間を定常的に繰り返すことであると

見なせる．理論的な構成要素は設計者の仮説あるいは考察中のシステムやモジュールがいかに動作すればよいかという概念から成っている．経験的な構成要素はシステムが実際にどのように動作しているかを測定することである．測定を行なうことによって設計者は理解を変え，さらに測定が必要なことを感じる．

測定および結果として得られた数値の解釈は難しい仕事である．すなわち，もし設計者がそのシステムがいかに働くかについて明確な理解をしていなければ誤まった結論にしか到達しない．またどうにかして数値を得ることは簡単であるが，これらの数値から洞察を得るには注意深い思考と計画が要求される．理論的あるいは概念的な理解や測定だけでは十分ではないということはあまり強く強調できない．理論的な解析は，存在しない問題あるいは重要でない問題を解くことがあるし，また重要な問題を見落とすこともある[3]．何を測定し，何が結果として予測されるかを理解しないで測定することは重要なパラメータを見落とすか，意味のない数値を得ることになる．

タイムシェアリングやその他のマルチプログラミング・システムの複雑性は測定および評価のための道具を開発する上で大きな努力の必要性を生じさせている．過去の設計者達は設計過程において構成要素向きの素朴な道具にあまりにも頼り過ぎる傾向があった．設計者の判断の多くは正当性を立証したり十分に評価することが殆んどできなかった．システム設計者の要求に見合う測定および評価の技術について述べた論文が発表され出したのは1960年代の半ば以後からである．システムの設計や応用はそれらの十分な理解や評価を越えているように思われるため，さらに多くの努力が必要とされる．本章ではこの重要な分野に現われる概念や経験のいくつかについて概説する．

最初の原則は測定および評価は設計の初期段階から販売，構成，設置および再構成の段階まで，そのシステムの一生を通して行なわれるべきであるということである．設計の初期段階からシステムの中には各段階で必要となる測定のための“ホック”を設計しておく必要がある．そうすれば測定と評価はシステムのライフ・サイクルにおける無理のない自然の一部となる．システム内にこれらの測定のためのホックを置くために，設計者は設計目標および設計がこれらの目標をどのように達成するのかについての理解を十分明確にしておく必要がある．この過程は初期の段階からシステムの中に測定および評価技術を設計する際の大きな間接的な恩恵をもたらす．すなわち，より良い理解が得られる．これは余分な努力から出てくるが，測定すべきものを定義するには明白かつ厳格に考える努力が要求される．それは設計者はモジュールやシステムが各

段階で実際どのように動作するか理解していなければならないことを意味する．この過程からシステムをさらに性能よくできるという直接的な恩恵が得られる．

測定技術はハードウェアおよびソフトウェア技術の2つに有効に分けることができる．ハードウェア技術は測定に使うことのできる標準的な機能と，特殊なハードウェア計測技術に分類できる．ソフトウェア技術はシミュレーションおよび解析的なモデル，あるいはシステム内の測定処理とに分類できる．

7.1.2 ハードウェアによる測定

初期のシステムでは標準的なハードウェア機構がプログラムのデバッグを簡単にするために組み込まれている．IBM-650はコンソール・スイッチからアドレス・ブレーク・ポイントをセットできるようになっていた．計算機はこのアドレスに達するまで実行しそこで停止する．それからプログラマは解析のためにレジスタを調べたり，記憶をダンプすることができた．IBM-704以後のシステムには**割出し・飛越しモード**(trapping-transfer mode)があり，そのモードでは飛越し命令が出てくる度に計算機は自分自身へ割込み，その事象を記録するユーザ・プログラムの固定アドレスに飛び越す．同じような機構が現用の計算機にもあるが，一般に保守パネルからのみ操作できる．タイムシェアリング・システムにおけるオンライン・デバッグ・システムにはソフトウェアで同じ機構がある．

これらの機構はシステムの測定にはあまり有用ではない．上述の最初の機構は手操作であり，2番目のものは著しく実行時間を増やす．仮に時間料金を喜んで払うとしても，この計算時間の増加はシステムのタイミングを変え，測定を意味のないものにしてしまう心配がある．高度に時間に依存するタイムシェアリング・システムではCPUタイミング，周辺装置タイミング，遠隔端末タイミング，補助記憶のタイミングに相互作用があるので，このようなことは十分起こり得る．CPUのような構成要素の1つを遅くすると，システムを大きく乱すことがある．ここで理解すべき概念は，もし得られる数値を意味のあるものにするには，その測定技術がシステムを大きく乱すことがないようにすべきであるということである．“大きく乱す”ということが何を意味するかは厳格には定義できないが，設計者は潜在的な危険性に気付いていなければならない．

標準的な機構は測定にはあまり有用ではないということがわかっているので，設計者は特殊な装置に頼っている．それは概念的には非常に単純ではある

が，その応用と結果の評価には熟練が必要なようである．代表的なハードウェア測定機器の1つはＩＢＭの**システム性能記録装置**（systems performance activity recorder）（SPAR）[5] である．これは論理パルスや論理レベルを知るために回路に接続できるオシロスコープで使われるような高インピーダンス・プローブ256本：カウンタ数個：クロック：AND，OR，補助ゲート，フリップ・フロップ，および比較回路のような基本的論理要素：およびプラグ・ボードから成り，カウンタの内容を磁気テープに記録する機能がある．

プローブはアドレス線，命令カウンタ，モード・フリップ・フロップ，その他の制御および情報線に接続できる．プローブ信号の多くはクロック信号と結合され，事象の測定を行なうようにプログラムされたプラグボードに従って論理要素を通過する．事象を示す信号はカウンタに入れられる．カウンタをテープに定期的にダンプすることもできるし，事象制御の下でテープにダンプすることもできる．UCLA の SNUPER システムは上述の方法を一段階進めたものである．それは事象信号とカウンタ類を直接汎用ディジタル計算機に接続している[4]．計算機は収集機器に出力することにより，その論理に影響を与えることができる．

興味ある事象をいかに測定するかを知るために次の例を考えよう．CPU，チャネル，あるいはディスクといった装置が使われる時間の割合を知りたいと仮定する．一般にはその装置が動作しているときは1の論理レベルを，休止している時は0の論理レベルを出力するフリップ・フロップがある．そこでプローブをこのフリップ・フロップの出力に接続して AND ゲートの1つの入力とし，同時にその AND ゲートのもう一方の入力にクロック信号を加えることによってその装置が動作中のときだけクロック・パルスの入力を得ることができる．ここでその AND ゲートの出力をカウンタに接続すれば装置動作中のクロック・パルス数を計数することができる．クロックを他のカウンタに直接接続すれば，基本時間が得られる．事象カウンタを基本時間カウンタで割ることにより測定期間中の利用率がわかる．図 7.1 にこの例を示す．

別の例を考えてみよう．システムがある特定のルーチンにどの位の割合で時間を費しているかを知りたいとする．ルーチンの開始アドレスおよび終了アドレスはわかっている．記憶装置のアドレス線に数本のプローブを取付ける．これらの信号を論理要素に通して，プロセスがそのルーチンから出たことを示す信号を発生させる．前者の信号は自走カウンタを動作させ，後者の信号はそのカウンタを停止させるように使う．もう1つのカウンタを基本時間を記録するために使う．このように構成すれば必要な測定ができる．もっと凝った測定も

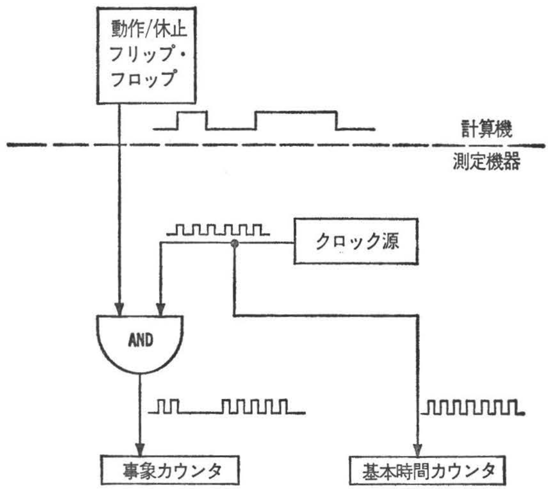


図 7.1 CPU 利用率を測定するための測定機器

できるが，ここで挙げた例は基本的な原理を示すものである．

特定の実験で得られるデータはテープ上に収集され，後で計算機によって処理され，出力解析結果が印字される．データの整理編集原理および理論と測定の相互関係はハードウェア，ソフトウェア技術の双方に共通であるので，この事項に関しては 7.1.3 節に譲る．

このハードウェア測定技術のはっきりした利点は，(1) 実際にシステムへの干渉を起こさない，(2) CPU 利用率のような比較的大きな測定のみならず，マイクロ秒あるいはそれ以下のレベルの非常に細かい測定ができる，(3) システムのあらゆる部分にアクセスできる，ということである．この技術はハードウェアおよびソフトウェア双方の性能を測定するのに使うことができる．

ハードウェア測定技術のはっきりした欠点は，(1) 実験はハードウェアの知識に基いて注意深く計画されなくてはならない，(2) プローブ，カウンタ，および論理要素の数が限られているので，一時に限られた数の実験しか行なえない，(3) プラグボードのプログラムは念入りにデバッグされなくてはならない，(4) 測定を行なうためにはハードウェアに知識を持つ人が必要である，(5) 原因と結果の関係を得るのが難しい，(6) 各実験を準備するのにかなりの時間を必要とする，ということである．次節では種々の測定に適したソフトウェア技術について考える．ソフトウェア技術は初めからシステムに組み込まれていれ

ば，特に簡単に使うことができるように思える．ハードウェア技術は設計の段階でソフトウェアの中に測定用ホックを置かなかったシステムに対してはある限られた目的にうまく使える．またハードウェア技術はマイクロ秒レベルの非常に細かいサンプリング，あるいはプログラムがアクセスできない測定事象に対してさらに有用である．

7.1.3　ソフトウェア技術

シミュレーション　タイムシェアリング・システムの設計におけるシミュレーションの有効性については，この分野における経験もまだあまりないので，いろいろ論争のある問題である．特に初期のある誤った経験により多くの誤解があるように思われるが，これはおそらく一様に細かいレベルでシステム全体をモデル化しようとしたシミュレーションから出て来たものであろう．一般に大きなシミュレーション・モデルを書いたりデバッグする方がシステムの骨格を書くよりもずっと長い時間がかかるという結果となっている．それゆえ，シミュレータの手助けで解決した問題はそれらが持ち上がる時点で解くことができない．

設計者は得られたデータを受入れるためにシミュレーション・モデルの正当性について自信を持たなければならない．通常，シミュレーションを書く人と設計者とは別人である．この事実ならびに設計者がそのシミュレータが確かにシステムの動作中の状態を反映しているかどうかを知る方法を持ち合わせていなかったことがモデルに対する信頼を失わせた．そして設計者はシミュレータとして概略のシステムを使う方を選んだ．

考察している問題に関係しているシミュレーション・モデルを多くのレベルの詳細さで作り出すことができる．特定の種類に対する問題に答えるために多くのシステム構成要素に対しては大きなレベルの詳細さを用い，1つかそれ以上の構成要素だけについては細かいレベルの詳細さを用いて有用なモデルを作ることができる．たとえば，Scherrはむしろ粗いモデルによって有用な結果を得ている[6]．General Electric での経験はシミュレーションが非常に有効な方法であることを示した[2]．そこでは設計者自身がタイムシェアリングの端末にすわり，ある特定の決定を行なうために簡単なシミュレーションができるようになっている．シミュレーションから得られた結果は時間的にも時を得ており，設計者自身によって行なわれたものであるので信頼があり，実際に使われた．Stanford における Nielson のモデルは Stanford の IBM 360/67 を構成する上で，また潜在的な障害を取除く上で極めて有益であった[7,8]．Oppenheimer

と Weizer は種々のスケジューリングおよびページング・アルゴリズムを評価するために簡単なシミュレーション・モデルを使った[9]. Fine と McIsaac もまたスケジューリング・アルゴリズムを評価し，パラメータを選択するためにシミュレーション・モデルを使用した．その結果は SDC システムの設計に使用された[12,13]. Pirtle は記憶装置の設計と記憶バスの構成を研究するためにシミュレーションをうまく利用した[10]. この分野の経験が増えるにつれてシミュレーションの巧妙な使い方が報告されてきている[27].

帰着する結論はシミュレーションはシステムの設計および構成の試案で有用な道具となり得るが，他の道具と同様に区別して使われなければならないということである．計算機システムのシミュレーションをどのように設計するかについての詳細に立ち入ることは本書の範囲を越えることであるが，いくつかの意見を述べることができよう．シミュレータを動作させるためにはデータが必要である．システムが既に存在している場合は種々のパラメータについて測定が行なわれ，これらのデータは動作中のシステムの変更を研究するのに使用される．システムがまだ存在していない場合には他のシステムで収集されたデータを設計中のシステムに適当に合わせて使うことができるし[11], また標準的な確率分布を使うことができる．有用なデータの形式には[6], (1) 種々の機能に対するコンソールにおけるユーザの思考時間の分布，(2) プロセスの記憶要求の分布，(3) 要求のサービスに必要なプロセッサ時間の分布，(4) 端末 I/O あるいはその他の I/O のためのブロックが生じる時間々隔の分布，(5) 記憶共用に対する要求の分布がある．要求される精密なデータはそのシステムとシミュレーションの設計に依存する．シミュレーションはプロセスに対してこれらの分布によるパラメータを適用し，そのモデルへの入力としてこれらのデータを使える．モデルの出力は応答時間，装置の利用率などに関する情報である．データに関して重要なことは，比較的簡単なモデルは短時間に開発でき，特定の問題に対して有用な解を与えるということであろう．問題ごとに別々のモデルが必要である．

解析モデル　計算機システム設計に関する色々な見方を研究するたの解析法を開発することには多くの感心が集まっている[6,15-21,28]. 関連している数学は研究中の変数の代数的関係を決める程度の簡単なものから，待ち行列理論，数学的プログラミング，マルコフ過程理論のようなランダム過程を研究するための複雑な技術を研究するものもある．これらについては本書の範囲外である．

ランダム過程のモデルに対しても色々な論議が行なわれている．ランダム・

プロセスの解析モデルには一般にそのデータ入力として平均値が使われる．通常，研究中のシステムに対して数学的に扱いやすくするために多くの簡単化の仮定が必要である．実データの分布形態によって平均値を使うことがしばしばひどい結果をもたらす．同様に必要とされる多くの簡単化の仮定は多くの興味ある問題の研究を不可能にする．解析モデルは主に大きな現象や一般的な効果を研究するのに有用であると思われる．

実際の計算機システム設計において解析モデルの技術を使うことは未だシミュレーション・モデルを使った場合のような成功は収めていない．1つの理由は簡単化のために必要な仮定の数であり，これはそのモデルに対する信頼を減らしていることと，別の理由は関連した数学的技法に対する広範囲な知識が不足していることである．実際の設計の場面で解析技術を利用する事はより経験を積むにしたがって増えてくるだろう．

システム内部の測定　システム内部の測定技術は主に事象計数，追跡技術，サンプリング技術という3つの形式に分類することができる．事象計数は単に特定の時間内にある事象が発生する回数を数えることである．追跡技術はある順序で生じる事象についてデータを記録する．サンプリング技術は周期的にシステムに割込み，レジスタ類，セル，テーブル，その他のデータ構造，ハードウェア・ユニットなどの状態を記録する．これらの技術に関する詳細を述べる前に問題解析における理論と測定の相互作用の例を示すことにしよう．

大部分の解析において興味ある問題は，時間はどこで費やされたかということである．いったん解が得られるとシステムに対する理論的な理解に基いてその解が正当かどうかを判断できる．もしその解が正当ではないと思われるなら費やされた時間配分の原因を判断するためにさらに厳密に調べる．たとえばCanrell と Ellison は各プログラムに対して開始および終了時間を与えた計数データをいかにして調べるかについて述べている[3]．疑問となる問題は，残りの時間はどこで費やされたかであった．

彼等はその時間はオーバーヘッドに費やされたと仮定した．そしてユザー・プログラムに対するものと同様な形式のシステム・モジュールのすべてに対するCPUおよび I/O の利用率に関して有効な統計を持っていた．彼等はこれらのデータから洗練された報告を作り出すプログラムを書こうと思えば書けたが，システム稼動状態が十分理解されるまで**初期の洗練された報告は時間の浪費にちがいない**という有益な原則にしたがってそうはしなかった．それゆえ，単にシステム利用率の統計を含むテーブル類のコア・ダンプを行ない，それら

を入念に調べた．その結果，オーバーヘッドは極端であるとは思われなかったが，彼等をもっと驚かしたのはシステムに大きな負荷がかかっていると想定したにもかかわらず，多くの休止時間があることが分かったことである．このように洗練した報告は必要ではなかった．

計数して得られるデータではなぜ大きな休止時間があるのかを決定するには十分ではないので，オペレーティング・システムを追跡モードで走らせ，その時に主記憶を割付けるスケジューリング・アルゴリズムにはいるような重要な事象を追跡収集プログラムで捕え，各事象に時刻を印した．これらのデータは一時貯えられ，最後にテープに書き出された．5分の稼動で得られたデータは総計350,000項目となり，データは整理編集されて10,000項目がプリントされ，それらに対して研究が進められた．各段階で動作の理論と実際の測定結果が比較された．理論およびデータの整理編集方法については各段階で修正された．

これは非常に大変な仕事であったが，結局は2つの相補的な測定方法が得られ，これらは他の状況の問題に適合でき，元の問題に答えたのと同様に多くの分からなかった性能上の障害を発見した．

上の例から言える重要な教訓は，(1) 理論と測定の相互作用，および(2) 初期段階でいかにデータを解析し整理編集するかについての先入観を避けることの有効性，についてである．オペレーティング・システムの改善を行なった期間中に彼等は大体14の性能上の障害を発見し，その修正を行なった結果，カストマ側でジョブ・ミックスにより10から50パーセントの性能改善が得られた．

いろいろな場合に興味がある解析報告の種々の形式の例を図 7.2に示すが，これは Estrin の報告[4]にしたがってモデル化したものである．これらの報告は以下で述べるハードウェアあるいはソフトウェアの測定技術で収集されるデータから得られる．

ここで測定技術についての議論へ戻ることにする．初期段階からシステムの中に測定技術を組み込むことは，開発者がそれらを容易に利用でき，また興味あるパラメータや統計が容易に測定できることを保証するために重要である．たとえば種々の待ち行列におけるプロセスの待ち時間を知ることは重要である．もし測定技術システム内に組み込まれたものでなければこの情報を得ることは困難である．その他の例として，ある与えられた時間にどのプロセスが主記憶に存在するかを知りたいことがある．この情報はシステム内では常に得られるが，もしこの統計を収集する方法が最初からシステム内に設計されていない場合は，後から答を得るためにはいくつかのテーブルをアクセスするような方法を採らなければならない．このアクセスはシステムに対する重大な妨害と

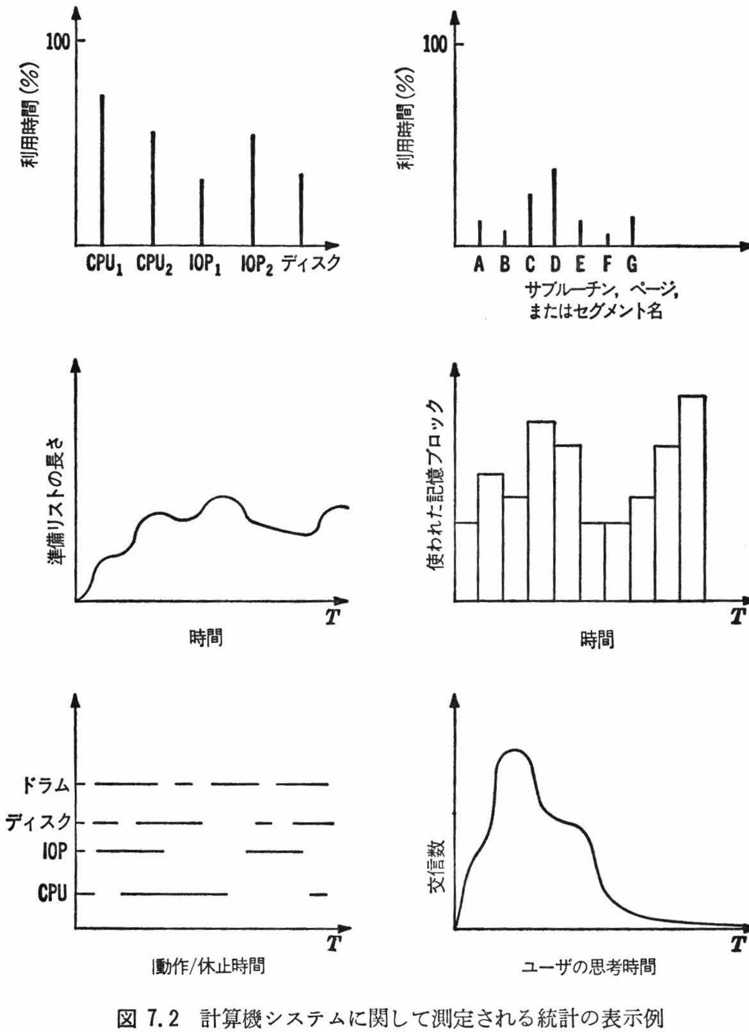


図 7.2　計算機システムに関して測定される統計の表示例

なり，情報の収集を無意味なものとし，パラメータは事実上測定できなくなる．それゆえ，重大な妨害を与えず別個に測定および研究ができるシステムに各システム機能を設計することが重要であると思われる．ここで3つの測定形式について考えることにしよう．

事象計数 (event counting) は単純であるが有用な技術である．あるシステム機能にはいると，ある記憶場所が増される．これによりスワップ回数，デ

ィスク・アクセス回数，あるいは興味あるどのような事象の発生頻度に対しても研究できる．たとえばCampbellとHeffnerはシステム・プログラムの優先順位が不適当に設定されていたたために非常に多くのスワップが生じているのを発見した[2]．事象計数を行なわなければおそらく発見できなかったであろう．なぜならば性能が低下する以外は外部的な症状がなかったからである．これらの事象カウンタは定期的にディスクあるいはテープに書き出され，リセットされる．

事象追跡（event tracing）は論理デバッグと性能測定および改善に有効である．その基本的な概念は事象に対するコードとその事象が発生した時刻といったデータを記録することである．CampbellとHeffnerはGEのGECOS IIIオペレーティング・システムで追跡したほぼ50の事象を記録した[2]．各事象は事象コードを含む2語のデータに記録された．事象別に収集されたデータの例としては，時刻，待ち行列の長さ，ジョブ識別子，機器番号，プロセス番号などがある．

各事象は追跡することもまた追跡をやめることもできる．追跡されたデータは最も最近のN個の事象を記録する大きな循環テーブルに集められる．後の解析のために追跡した事象をテープに書き出すための準備も行なわれる．追跡機構は新たに追跡を加えたり，ある事象に対して別のデータを収集できるように統一した方法で実現されなければならない．事象追跡は多量のデータを生み出すことがある．解析の目的のためには追跡情報から得られる事象の計数，連続事象間の経過時間，同じような事象間の経過時間，ジョブ識別子あるいは他のパラメータによる分類といった形で報告を作成できる．考えている問題にしたがってその他にも種々の形式の報告が考えられる．

サンプリング技術（sampling technique）は使いやすく強力であり，システムおよびユーザ・プログラムの両方に適用できる．必要なものはインタバル・タイマだけである．時間々隔を細かくできればできるほど，システムの振舞に関する研究をより詳細に行なうことができる．理論的には割込みはランダムあるいは一定間隔で起こるであろうが，この時間々隔は研究中のプログラムに存在するいかなるパターンにも統計的に無関係であることが知られている．実際のタイムシェアリング・システムではプログラムとインタバル・タイマの割込みは同一時刻に生じることはない．これはプログラムの実行時間が他のプログラムとサービスを取り合う状況からランダムな性質となるからである．もし同一時刻に起こるならばサンプリング間隔を加減すればよい．

サンプリング時にシステムにはいり，ハードウェア・ユニット，待ち行列，

あるいはその他の状態情報を記録する．システムとユーザ・プログラムの両方に対してプログラム・カウンタの内容を記録できる．プログラム・カウンタの与えられた命令シークエンス内に存在する頻度がそのシークエンスに使われる全時間に比例するという考えを用いて，プログラムの効率を上げるために特別の努力を払う価値のあるプログラムの障害や領域を取り除くことができる．収集した多量のデータ解析し，プログラムの種々の部分で費やされた時間のヒストグラムを得るために，データ整理編集プログラムが必要である．解析プログラムには一般に記号テーブルを利用できないので，ヒストグラムは数値アドレスの形になる．したがって，ユーザはコンパイル・リストで与えられたコア・マップの結果を関係付けなければならない．

この議論はシステムあるいはユーザのプログラム時間の分布に関する解析を取扱ったものである．ハードウェア・ユニットの利用率，ディスクのシーク時間，待ち行列の長さ，あるいはサンプル時に収集されたその他のいかなるパラメータに関しても同じような解析ができる．サンプリング技術を使うことによるシステムへの妨害は，サンプリングの間隔と各点で収集されたデータ量に依存している．

7.1.4 まとめ

測定および解析に対する動機づけを示し，設計者は設計の初期段階からシステム内に測定機能を含めるように力説してきた．評価する際の理論および測定段階の重要性を強調した．データの収集は容易であるがその意味を解釈することはしばしば難しい．収集すべき統計に関して有益な疑問は，その統計によってどのような決定が影響されるかということである．このような考え方はシミュレーションの必要性を問う場合にも役立つ．シミュレーションあるいは測定で得られる形式の模擬データを作り上げ，それらが意味深長に使えるかどうかを調べることが有用な訓練になる．また同様に，計算機およびデータを出力する機器で作られる報告の書式も注意深く考慮しなければならない．粗い走査にはコア・ダンプで十分であろうが，時にはより洗練された報告が必要である．報告としてテーブルを打ち出すことが適当な時もあるし，ヒストグラフや図形的な出力をプリントすることがもっと有効なこともある．適当な表示書式でオンラインCRTを使うと設計者は研究中のパラメータの動的な様子に対してより良く感覚を得ることができる．またシステムが最終的に完成し引き渡される時に計算機のオペレータにとってより便利なものになる．

ハードウェアおよびソフトウェアの概念に関して色々な検討を行なったが，

これらはあらゆるシステム設計者が蓄積した技術の重要な部分であるように思われる．これらの技術は色々な立場でより頻繁に現実のシステムに現われるであろう．

7.2 信頼性，保守性，および回復性

7.2.1 概　　要

タイムシェアリング・システムは信頼性に富み，保守が容易で故障が生じた時に段階的に回復できるのが当然であると言うのはまったく明らかなことを述べている．しかしながらこれらの目標を達成する上で遭遇するいくつかの問題点と方法を簡単に検討することは有用である．バッチ処理に使われた初期のシステムが誤動作を起こしたときには，ユーザにはほとんど損害は与えず，ただ仕事を戻すのが遅れただけであった．一方，タイムシェアリング・システムではユーザがコンソールにすわってかなりの努力を払っており，システムの故障が生じるとそれらが無駄になる恐れがある．ユーザはスケジュールさえされればシステムは動くものと思っているし，そのファイル・システムは完全に動作すると期待している．なぜならば，ユーザ・ファイルがユーザのコンソールでの努力を表わしているからである．万一，故障が生じた場合には故障した時点から再開できるような方法でシステムを回復してくれるものと思っている．

故障にはハードウェアの故障とソフトウェアの故障の2つの形式がある．タイムシェアリング・システム（Multics のような特別に大規模なもの）ではバッチ・システムに比べてかなり多くのハードウェアがあり，そのため信頼性を得るのはより困難になる．タイムシェアリング・システムは補助記憶装置やその他の機器を他のシステムより頻繁に動作させるので，その余裕をもたせた素子や設計が必ず困難さを作り出している．ハードウェアとソフトウェア・システムの非常に密接なつながり合いはその問題に対する責任がどこにあるか決めるのを難しくしている．ソフトウェア・システムは複雑で，通常，数人の手で設計され書かれるので，使用に供する前にすべての条件下で検査するのが難しい．

これらの事実はかなりの開発と検査に対する経験を積んだ後でも必ず故障が起こるという仮定でタイムシェアリング・システムを設計しなければならないことを意味している．誤動作が存在する中で信頼性を増し，故障の源を分離するのを助け，そしてまた故障回復をユーザへの干渉を最少にして円滑に行なう

ために，ハードウェアおよびソフトウェア双方の技術を設計の初期段階からシステムの中に盛り込むようにすべきである．この分野の研究は難しいが急速に発展している．本節ではこの発展している有用な概念のいくつかを述べる．

7.2.2　ハードウェアの信頼性

対処しなければならないハードウェアの故障にはプロセッサあるいは記憶装置の誤動作，電源異常あるいは瞬時停電，直接アクセス機器上のデータの読み損い，および周辺装置の誤動作などがある．信頼性を上げる最良の方法は明らかに必要な機能を遂行する最も簡潔な論理装置を設計することである．設計はいかなる電気的あるいは機械的な構成要素も過負荷点の近くで動作しないように保証した最悪条件で行なわれる．あるモジュールにおける故障が他のモジュールへ広がり得る電気的あるいは機械的条件を作らないように，また故障したモジュールをシステムから容易に取り除けるように，モジュールに適当なインタフェースを設けることは実用的な良い手段である．

主記憶，CPU，補助記憶，および周辺端末間でのデータ転送を検査するには誤り検出・訂正コードを用いるのが常識である[23]．演算操作で保存される符号もまた演算の変換機能を検査するのに使われる[23]．誤りの検査は故障ユニットの位置を示し，故障をマスクしたり故障からの回復に使うハードウェアおよびソフトウェア技術を可能にする上で有用である．マスク技術の例はソフトウェアあるいはハードウェアの解析で再試行が可能であると判断した後で行なう命令またはデータ転送再試行である[23,24]．

信頼性を向上させる一般的な方法はハードウェアの冗長を利用することであり，これは論理ゲートを2重にすることから主要な機能ユニットを2重にすることまでさまざまな設計レベルで適用することができる．後者の方法は商用計算機では最も一般的である．図 3.4 に示す Multics システムはそのような方法の一例である．このシステムでは I/O 装置から2台のプロセッサに2重の情報経路がある．2台のプロセッサともシステム機能を取扱うことができる．2重システムの性能改善は良く知られていないが，理論的に可能な100パーセント近いところにもいかない．性能改善の制約になるのは多重プロセッサの相互干渉を避るために設定されるハードウェアおよびソフトウェアのインタロックである．しかしながら，一般に性能改善が2重システムを作る動機にはなっていない．すなわち故障が起たった場合にかたわになったシステムが縮退したレベルでサービスを続けられるようにするという発想に基づいている．

IBM 360/67 および GE-645 の単一プロセッサ・システムに組み込まれた2

重化の別の形式が補助記憶およびいくつかの周辺機器への多重データ経路にみられる．たとえば図 3.4 を各 GIOC から何台の機器へ2重経路を示すように拡張できよう．仮に1台の GIOC が故障しても機器へは他の経路が存在する．同様に各機器は2台の制御装置を持っている．システム・モジュールが故障した場合，オペレータまたはシステムが適当なシステム・テーブルおよびスイッチを変えてシステムを再構成できる．その後，故障した機器を修理できる．故障装置を取り除くこの考えは記憶バンクにも適用できる．

記憶保護はユーザ・プロセスからのシステム保護と同様にシステム自身からの保護にも有用である．システムの多くの部分を読み出し専用あるいは実行専用として構成することにより，誤りを早く分離したり損害を制限できる．

通常の動作中にハードウェア検査を行なえる回路や特殊な保守用パネルを利用することが現在一般的になりつつある．故障素子の位置付けを手助けする特別な検査回路は設計の際に重要である[23]． これらの技術はハードウェアの信頼を向上させるために存在し，効果的にするには設計の初期段階から考慮しなければならない．

もう1つのハードウェア要素について述べる必要がある．すなわち電源と周囲条件である．電源の信頼性は無論のことと思われるがちであるが，初期システムの設計者ならびに設置の際には十分注意を払う必要がある[22]． 電力会社は高信頼性を得るために冗長な電源および線路を用いるが，電気的な故障および故障を直すための線路の切換えは瞬時的な停電を起こし高速回路に影響を与える．普通の安定化電源ではサージ・スパイク，周波数変動，およびサイクル欠損に対処するには必ずしも十分ではない．これらは直ちに高速回路に影響を与える．円滑な電源を得るために特別な電動機一発電機・バッテリー・システムを使うのが一般的になってきた．

温度および湿度に対する回路の感度もまた信頼性の問題になる．空調装置が故障した場合，システム・キャビネット内に温度および湿度の感知素子を置いて危険に近づいた時に警告を出すようにする．

7.2.3 ソフトウェアの信頼性

概　要　信頼性を上げ，円滑な回復を行なわせたり，保守する間にシステムへの干渉を軽減するためのソフトウェア技術はしばしば相互に関係するが，どちらかといえばシステムに依存する．しかしながら開発された技術を示すためにいくつかの概念と例を引出すことができよう．CPU，主記憶，およびシステム・プログラムの故障について考えてみる．補助記憶を使う技術に関して

は6.1.7節で述べた．

一般的な故障の形式にはループに落入ったり禁止された記憶領域で実行することなどがある．それだけでは重大ではない小さな故障でも大きくなる傾向がある．目標は損害を極所化し円滑に再始動できる可能性を最大にするように，できるだけ早く誤動作を検出することである．故障の影響は最小限度のユーザに留めるべきである．

この目標を達成する一方法は，特別な検査プログラムをシステム内の重要な部所に組込むことである．これから検討する技術はオーバーヘッドの増加は少ないように思えるし，信頼性ならびに誤り個所発見に利点があり，一考に値する．重要なシステム情報はテーブル，待ち行列，バッファ，および特別な記憶領域に含まれている．故障の発生時に円滑な回復を行ない，すべての誤りをマスクする能力を増強させようとするならば，これらの情報の保護を最優先にしなければならない．

あらゆる誤り検出・訂正コードあるいは技術の背後にある基礎的な要求事項は情報の冗長である．システムには慣例，構造，あるいは実現化における相互関係によって冗長が暗黙のうちに含まれることがあるが，検査する目的で明白に冗長を持たせることもある．設計者はこの双方の冗長を使うことができる．これらの特長を示すいくつかの常識的な技術について述べる．一般に検査を行なう上で最も効果的な点はシステム・テーブルのアクセスや更新を行なうルーチンの入口である．これは検査を容易に実現しようとするならば，別々のシステム・ルーチンでテーブルを管理するより，テーブルのすべてのアクセスあるいは更新を共通のルーチンを通して行なうべきであることを意味する．

検査のために暗黙のうちに冗長を含めた例　誤りの検出および訂正のために暗黙のうちに冗長を含めた一例を考えてみる．多くのシステムはリスト形式でテーブルや待ち行列を持っている．特殊な例は5.2.6節で述べたXDS-940システムのスケジューリング待ち行列の構造である．ハードウェアの誤りや誤動作中のプログラムがこれらのリストを解いてしまったり，あるいは小さなループにしてしまうことはよくある．前者の場合にはリスト上を動くスケジューリング・アルゴリズムが別のコードにそれて混乱を引き起こす．また後者の場合には少数のプロセスのみが十分なサービスを受け，残りのプロセスは実質的にシステムから分離されてしまうことになる．この場合，重大な誤動作を引き起こす可能性がある．

システムが稼動しているとき，そして初期のデバッグ段階でさえも，いかに

リストが不当に解かれたりあるいは縫い合わされたかはわからず，誤動作しているハードウェアの原因あるいはプログラム・モジュールを取り除くことは困難である．不可解な結果はスケジューリング・アルゴリズムが故障点に達するまでいくつかのプロセスを正しく実行しながら待ち行列をゆっくり動かすことから起こる．故障時点では多くのテーブル類や記憶領域は書き換えられ，多くのプロセスが実行されてしまっているので，その原因についての手掛りはほとんど残っていない．そしてあまりにも損害が大き過ぎて円滑な回復はできない．

この問題を縮少し分離するのに使える技術はリストを操作する部分に検査ルーチンを置くことである．このルーチンはリストの挿入，消去時にそれがどのように縫い合わされているかを調べ，したがって操作後にリストがどのようになるかを知ることができる．この検査ルーチンがリストのある一要素が空欄あるいは誤った要素を指しているのを発見するとある種の報告がなされ，訂正動作が行なわれる．

2つの明らかな誤り発生源が問題を作り出している．すなわちリスト・ポインタ内のアドレス・ビットに影響を与えるハードウェア故障と呼出しプログラムがリスト処理ルーチンへ渡した不当なポインタである．

これらの誤り源を検査するのに使える冗長情報がある．XDS-940 システムでは，すべての待ち行列のエントリーは特定の記憶領域内のセルのブロックになっている．したがって正しいリスト・ポインタはすべてこの領域内を指している．さらに各ブロック内のリスト・ポインタはそのブロックの同一相対記憶場所を示している．この情報はリスト変更の基準として使われるリスト・ポインタの正当性を検査することにも使える．この形式の冗長情報は強力な誤り検出能力を与える．検出に関する詳細なことは全くシステムに依存するが，おそらくその原則は色々なシステムに広く適用できよう．

誤りが検出できたと仮定すると，XDS-940 では次の手順で回復される．もちろん誤りは処理する前に誤ったポインタを渡した呼出しルーチンに関する適当な情報とともに記録されているものとする．待ち行列の要素は決して物理的には移動されず，ただポインタを変えることによって待ち行列の別の位置に置かれるだけである．それゆえシステムはすべての要素がどこにあるかを知っている．たとえ待ち行列が分解されても，システムは待ち行列エントリーを含む領域の中を通り，リストを元通りに縫い合わすことができる．前に操作されていた要素への正しいポインタは呼出しルーチンあるいは適当なシステム・コミュニケーション・セルに戻される．最悪の場合には実行中のプロセスは終結さ

れ，このことを示すメッセージが対応するユーザの端末に送られる．その他のプロセスは影響されない．

I/O 機器の割付けに用いられるテーブルでその他の検査もできる．単に機器が利用可能であるかを検査してから処理するのではなく，まずその機器が空いているかどうかを知るための I/O 状態検査を行なう．このようにて冗長情報を得ている．もしテーブルがその機器は使用中でない事を示している時にそれが使用中ならば，どこかで誤動作が生じている．

検査のために明白な冗長を含めた例　**チェック・サム**（check sum）の概念は種々のデータ処理で検査の目的に使われ，またシステム・テーブルの検査にも使える．基本的な考えは次の通りである．テーブルの内容は合計され，この合計はテーブルと一緒に格納される．テーブル内のエントリーが変更される場合には，そのチェック・サムの予期した変化を計算する，次いでテーブルを変更し，その新しいチェック・サムを計算する．新しいチェック・サムと予期したチェック・サムを比較し，もしそれからが異なっていればテーブル内に誤りがあるか，あるいはチェック・サムを計算する際に誤りが生じたのである．もし元の情報が保存されていれば，瞬時的な故障と仮定して上の過程を繰り返せば良い．再試行の後でも誤りが残っていれば，その旨が報告され，適当な動作が試みられてそれ以上損害を与えないようにする．

チェック・サム法を使う上での秘決は予期したチェック・サムを計算することである．もしテーブル内の1ビットが変更される場合は，そのビット位置および1か0のどちらかにセットされるにしたがってチェック・サムが2の累乗だけ増加あるいは減少するはずである．この情報は予期したチェック・サムを計算するにも使える．テーブル内の1語全部を入れ換える場合，新しい内容から古い内容の値を引き，予期したチェック・サムを求めるためにその結果を古いチェック・サムに加える．もし1語のあるフィールドを入れ換える場合は変更後のエントリーの写しを作れる．この新しいエントリーは古いエントリーと論理的に比較され，2つの語の変更しないフィールドがすべて同じであるかどうかが調べられる．その後は1語全体を入れ換えるところで述べたチェック・サムの方法が使われる．

計算や記憶参照はあまり必要なくただテーブルへ行って変更するだけのためになぜこのようなめんどうな検査処理をするのだろうかという疑問が読者にはあろう．その答は正しいテーブルの内容がシステムの適切な動作には極めて重要であり，設計者は検査に金をかけてもやりがいがあると考えているからであ

る．チェック・サム技術は特に瞬時的なハードウェア誤り，あるいは記憶ビットの誤動作を監視するのに有用である．またその技術は誤ったビット，フィールド，あるいは語への格納といった大きなソフトウェアの誤りも捕えることができる．チェック・サム技術はシステム内のどこにも，たとえばシステム・モジュールやスワップされるファイルのようにも使える．ファイルあるいはルーチンが主記憶内に入った後で計算されたチェック・サムは，そのファイルあるいはルーチンと一緒に保存されていたものと比較される．このような方法で新しい複写あるいはオーバーレィが正しいという高い可能性を得ている．新しい複写で置換えられるためにはモジュールはそれ自身で修正してはならないしデータを含んではならない．チェック・サム技術は補助記憶の割付けに用いる重要なテーブルで特に有用である．補助記憶で使われるその他の技術は正当な領域あるいはブロックの開始記憶場所を指すかどうかを調べるために発生されたアドレスを検査することである．

システムがいかに動作するかを理解している賢明な設計者はシステム内に暗黙に含まれた，あるいは設計者自身が明白に含めた冗長情報を用いて種々の検査法を考え出せる．システムが誤りを早く検出できればできるほど，より容易にその原因を発見しマスクすることができる．

回　復　回復手順の設計目標は故障点にできるだけ近い状態に戻すことによって影響を与えるユーザをできるだけ少なくすることである．簡単な回復方法は単にシステムの新しい複写をロードし初期設定をすることである．この方法はすべてのユーザに影響を与えるので上に述べた目標に反する．上に述べたように，いかなるユーザにも妨害を与えず仮に与えたとしても1人か数人にとどめることができるようなある種の訂正あるいは回復を行なうために十分早期に誤りを検出することは可能である．誤りが新しいシステムの複写を必要とするほど重大であると判断された場合は，円滑な回復を行なうために古いシステムから新しいシステムへテーブルを移し変えてみるべきである．

いくつかの XDS-940 システムで使われている回復手順は，空いている記憶領域あるいは補助記憶から再書き込み可能な読み出し専用情報を貯えている領域に回復プログラムをロードする．この回復ルーチンはすべてのユーザ・プロセスを記憶装置の外に出し，システムを一定の領域に入れなければならない場合は古いシステムをユーザ領域に移す．そうすればシステムの新しい複写をロードできる．次に回復プログラムは回復が簡単にできるかどうかを調べるために古いシステム・テーブルを検査する．この検査にシステム内に暗黙または明

白に含まれた冗長情報が用いられる．テーブルが損害を受けていないかあるいは修復可能であると判断された場合は，そのテーブルは新しいシステムに複写され，システムは再始動される．損傷を受けたテーブルのエントリーは少数のユーザに影響を与えるかもしれないが，ほとんどのユーザは正しく再始動できよう．再度述べるが，この手順がどのように行なわれるかは大きくシステムに依存するが，すべてのシステムに適用することができる．

ユーザ・ファイルが CPU，記憶装置，またはシステムの故障で傷付けられたと疑われるか，あるいはそれが明白になった場合は，適当な表示がそのユーザのファイル・ディレクトリに書かれる．そしてユーザがそのファイルをアクセスすると警告メッセージが打ち出される．同様に終結されたプロセスのユーザにもメッセージが出される．少なくとも可能ならば，ユーザにメッセージを送る試みをせずに通信回線を切断すべきではない．

保　守　本節では保守手順に有用な多くの概念について述べる．7.1.3 節で述べた事象追跡技術がシステムの保守にも役立つ．追跡は保守員にシステムの最近の計算経路に関する有用な情報を与え，故障原因を取除く際の手助けををする．

あるソフトウェア・モジュールが誤動作している時でもシステムを停止しなければならないということはない．故障したモジュールが提供していた機能はユーザ社会から取除かれるか，あるいはその故障が間欠的なものであれば正しいモジュールを用意するまでそのまま動作させておくことがある．一般にその準備はすべてのシステム設備を使っている端末から行なわれる．そのモジュールが用意されるとシステムの新しい複写が作られ，適当な時間にロードされる．

ハードウェア修復中に可能な時はいつでも同じ考え方を適用したい．一般に計算機は一組の診断プログラムが付属されて納められる．このプログラムはハードウェアを検査したり，保守員が問題個所を見つけ出すのに利用される．通常，これらのルーチンは計算機のすべての制御を使う．タイムシェアリング・システムではシステムの連続的な可用性 (availability) を得ることが目標である．単一プロセッサ・システムにおいては，一般にプロセッサが故障すると適当な修理を行なうまでシステムは利用できない．たとえ単一プロセッサ・システムでも色々な構成要素があり，それらが故障した場合にすべてのユーザがシステムから離れなければならないことはない．記憶ボックスあるいは重要でない I/O 装置の故障などは破滅的でない故障の例である．

そのような場合には，特別なユーザ・プログラムとして実行でき，故障の位置付けが可能で，システムを停止せずに修復ができるような診断プログラムが望まれる．永久的ではない間欠的な，あるいはデータに依存した故障の多くは取除くことが難しいが，もし診断ルーチンが特別のユーザ・プログラムとして実行可能であれば，そのためにシステムを停止する必要はない．テスト状態で保守員がプログラマにはアクセスできない回路を置き，しかも他のコンソールにいるユーザから見ると一般の稼動には妨害を与えないようなハードウェア設備を設けることが望ましい．

オンライン検査システムは概念的には極めて単純である[25]．それらはシステム・モードで実行しながら特別のルーチンにアクセスし，また必要な時にそれ自体を主記憶内に固定する能力を持つような一般のタイムシェアリングのプログラムとして書ける．そのシステムでは保守員に特別のライブラリ・ルーチンを提供すると同様に会話形言語を用意して診断プログラムが書けるようになっている．また保守員はある機器をサービスから外したり，それにテスト・データを送ったり，その状態レジスタをアクセスしたり，機器が起こす割込みを阻止したりできる．良い診断システムであるための条件の1つは，保守員がある種の機能をプログラム制御下においてオシロスコープで信号を追跡できることである．このような設備は計算機の全制御を使うわけではない．このようなシステムを書くには設計時に保守機能を考察する必要がある．たとえば，接続されている機器が適切に動作していな場合に I/O 命令を実行してもシステムに損傷を与えてはならない．XDS-940 における直接 I/O 命令はこの要求を満足していない．プロセッサはこの種の命令を実行すると先に進む前にアドレスされた機器からの確認信号を待つ．したがって，もしこの信号がこないと計算機は事実上永久的に停止することになる．それゆえ，直接 I/O 線のインタフェース回路および機器が動作していれば機器に対するオンライン・デバッグができる．もっと良い設計はプロセッサが直接 I/O 命令を出し，一定時間待って確認信号を受取らない時にはスキップするようにすることである．このためにはシステムを停止することなくインタフェースをオンラインで検査できなければならない．

7.2.4 まとめ

設計者はハードウェアおよびソフトウェアの誤動作が生じることを仮定しなければならず，したがって設計の初期段階からシステムに種々の機構を実現しなければならない．信頼性は注意深い設計，検査機能，および冗長によって向

上できる．冗長なハードウェア・ユニットあるいはサブユニットを用いたり，冗長コードを用いて，情報転送および演算の検査ができる．チェック・サムのような明白な冗長情報はソフトウェア・ルーチンで使われ，暗黙のうちに含まれた冗長情報はソフトウェア検査ルーチンで使われる．冗長技術をハードウェアで開発したりコーディングするさまざまな努力がなされており，多くの文献および経験を得ている．

ソフトウェア・システムで使う冗長技術の開発に注目しだしたのはつい最近のことである．システムには暗黙の冗長情報が多く含まれており，賢明な設計者はそれらを利用できるということを指摘した．円滑な回復を行なうにはシステム・テーブルを保護することが最も重要になる．効果的に冗長情報を利用するためにシステム・テーブルのアクセスは単一のルーチンを介して行なうべきである．

回復の目標は故障時にできるだけ近い状態に戻すことにある．故障によって影響を受けたプロセスやファイルのユーザには適切にその旨が知らされなければならない．

保守手順は可能な限りシステムの利用と並行して保守が行なえるように開発することが必要である．これはハードウェアおよびソフトウェアの保守の双方に対して言えることである．

本章で議論したシステム設計の要素はさらに研究および開発を行なう必要がある．

参考文献

1 **Calingaert, Peter:** System Performance Evaluation: Survey and Appraisal, *Commun. ACM,* vol. 10, no. 1, pp. 12–18, January, 1967.

2 **Campbell, D. J., and W. J. Heffner:** Measurement and Analysis of Large Operating Systems during System Development, *AFIPS Conf. Proc., Fall Joint Computer Conf.,* vol. 33, pp. 903–914, 1968.

3 **Cantrell, H. N., and A. L. Ellison:** Multiprogramming System Performance Measurements and Analysis, *AFIPS Conf. Proc., Spring Computer Conf.,* vol. 32, pp. 213–221, 1968.

4 **Estrin, G., et al.:** SNUPER COMPUTER: A Computer in Instrumentation Automation, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* pp. 645–656, 1967.

5 **Schulman, Franklin D.:** Hardware Measurement Device for IBM System/360 Time Sharing Evaluation, *Proc. ACM Natl. Conf. 1967,* pp. 103–109.

6 **Sherr, A. L.:** An Analysis of Time-shared Computer Systems, *MIT Project MAC Doc.* MAC-TR-18, Ph.D. thesis, June, 1965.

7 **Nielson, N. R.:** Computer Simulation of Computer System Performance, *Proc. ACM Natl. Conf. 1967*, pp. 581–590.

8 **Nielson, N. R.:** The Simulation of Time-sharing Systems, *Commun. ACM*, vol. 10, no. 7, pp. 397–412, July, 1967.

9 **Oppenheimer, G., and N. Weizer:** Resource Management for a Medium Scale Time Sharing Operating System, *Commun. ACM*, vol. 11, no. 5, p. 313, May, 1968.

10 **Pirtle, N. W.:** Intercommunication of Processors and Memory, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 621–633, 1967.

11 **Rehmann, Sandra L., and Sherbie G. Gangwere, Jr.:** A Simulation Study of Resource Management in a Time-sharing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1411–1430, 1968.

12 **Fine, G. H., and P. V. McIsaac:** Simulation of a Time-sharing System, *Management Sci.*, vol. 12, no. 6, pp. B180–B194, February, 1966.

13 **Schwartz, J. I., and C. Weissman:** The SDS Time-sharing System Revisited, *Proc. ACM Natl. Conf. 1967*, pp. 263–271.

14 **Hellerman, H.:** Some Principles of Time-sharing Scheduler Strategies, *IBM Systems J.*, vol. 8, no. 2, pp. 94–117, 1969.

15 **Belady, L. A., and C. J. Kuehner:** Dynamic Space-sharing in Computer Systems, *Commun. ACM*, vol. 12, no. 5, pp. 282–288, May, 1969.

16 **Belady, L. A.:** A Study of Replacement Algorithms for a Virtual Storage Computer, *IBM Systems J.*, vol. 5, no. 2, pp. 78–101, 1966.

17 **Babcock, J. D.:** A Brief Description of Privacy Measures in the RUSH Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 301–302, 1967.

18 **Fenichel, R. R., and A. J. Grossman:** An Analytic Model of Multiprogrammed Computing, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 717–721, 1969.

19 **Sherr, A. L.:** Analysis of Storage Performance and Dynamic Relocation Techniques, IBM doc. Tr-000-1494, 1967.

20 **Denning, Peter J.:** Effects of Scheduling on File Memory Operations, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 9–21, 1967.

21 **Denning, Peter J.:** The Working Set Model for Program Behavior, *Commun. ACM*, vol. 11, no. 5, May, 1968.

22 **Cheek, R. C.:** Fail-safe Power and Environmental Facilities for a Large Computer Installation, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 51–56, 1968.

23 **Conti, C. J., et al.:** Structural Aspects of the System/360, Model 85, *IBM Systems J.*, vol. 7, no. 1, pp. 2–29, 1968.

24 **Higgins, Alan N.:** Error Recovery through Programming, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 39–43, 1968.

25 **Nelson, G. W.:** OPTS-600: On-line Peripheral Test System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 45–50, 1968.

26 Oppenheimer, G., and K. P. Clancy: Considerations for Software Protection and Recovery from Hardware Failures in a Multi-access Multiprogramming, Single Processor System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 29–37, 1968.

27 Merikallio, Reino A.: Simulation Design of a Multiprocessing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, 1968.

28 McKinney, J. M.: A Survey of Analytical Time-sharing Models, *ACM Computing Surv.*, vol. 1, no. 2, pp. 105–116, June, 1969.

文 献 目 録

Amdahl, G.: Unpublished simulation study, IBM Corporation, 1965.

Anderson, G. B., et al.: Design of a Time-sharing System Allowing Interactive Graphics, *Proc. ACM Natl. Conf. 1968*, pp. 1–6.

Arden, B. W.: The Role of Programming in a Ph.D. Computer Science Program, *Commun. ACM*, vol. 12, no. 1, pp. 31–37, January, 1969.

——— **et al.:** Program and Addressing Structure in a Time-sharing Environment, *J. ACM*, pp. 1–16, January, 1966.

Babcock, J. C.: A Brief Description of Privacy Measures in the RUSH Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 301–302, 1967.

Badger, G. F., Jr., and E. A. Johnson: The Pitt Time-sharing System for the IBM System 360, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 1–16, 1968.

Belady, L. A.: A Study of Replacement Algorithms for a United Storage Computer, *IBM Systems J.*, vol. 5, no. 2, pp. 78–101, 1966.

——— **and C. J. Kuehner:** Dynamic Space-sharing in Computer Systems, *Commun. ACM*, vol. 12, no. 5, pp. 282–288, May, 1969.

Bell, G., and M. W. Pirtle: Time-sharing Bibliography, *Proc. IEEE*, vol. 54, no. 12, p. 1764, December, 1966.

Bobrow, D. G., and D. L. Murphy: Structure of a LISP System Using Two-level Storage, *Commun. ACM*, vol. 10, no. 3, pp. 155–159, March, 1967.

Bryan, G. E.: JOSS 20,000 Hours at the Console: A Statistical Survey, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 769–778, 1967.

———: Joss: User Scheduling and Resource Allocation, *The RAND Corporation, Mem.* RM-5216-PR, Santa Monica, January, 1967.

Calingaert, Peter: System Performance Evaluation: Survey and Appraisal, *Commun. ACM*, vol. 10, no. 1, pp. 12–18, January, 1967.

Campbell, D. R., and W. J. Heffner: Measurement and Analysis of Large Operating Systems during Development, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pp. 903–914, 1968.

Cantrell, H. N., and A. L. Ellison: Multiprogramming System Performance Measurement and Analysis, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, pp. 213–221, 1968.

Cheek, R. C.: Fail-safe Power and Environmental Facilities for a Large Computer Installation, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 51–56, 1968.

Coffman, E. G., and L. C. Varian: Further Experimental Data on the Behavior of Programs in a Paging Environment, *Commun. ACM*, vol. 11, no. 7, p. 471, July, 1968.

——— **and L. Kleinrock:** Computer Scheduling Methods and Their Counter Measures, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, p. 11, 1968

Comfort, W. T.: A Computing System Design for User Service, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 619–626, 1965.

Conti, C. J., et al.: Structural Aspects of the System/360, Model 85, *IBM Systems J.*, vol. 7, no. 1, pp. 2-29, 1968.

Corbato, F. J.: PL/I as a Tool For System Programming, *MIT Project MAC Mem.* MAC-M-378, July 2, 1968.

———: A Paging Experiment with the Multics System, *MIT Project MAC Mem.* MAC-M-384, July 8, 1968.

———: Sensitive Issues in the Design of Multi-use Systems, *MIT Proj. MAC Mem.* MAC-M-383, Dec. 12, 1968.

———: System Requirements for Multiple Access Timeshared Computers, Project MAC Document MAC-TR-3.

——— **et al.:** An Experimental Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 21, pp. 335–344, 1962.

——— **and V. A. Vyssotsky:** Introduction and Overview of the Multics System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 185–196, 1965.

——— **and J. H. Saltzer:** Some Considerations of Supervisor Program Design for Multiplexed Computer Systems, *IFIP Conf. Proc. Edinburgh, August*, 1968.

"CP/CMS Program Logic Manual," IBM Cambridge Scientific Center, Cambridge, Mass., 1969.

Curriculum 68, *Commun. ACM*, vol. 11, no. 3, pp. 151–197, March, 1968.

Daley, R. C., and J. B. Dennis: Virtual Memory Processes, and Sharing in MULTICS, *Commun. ACM*, vol. 11, no. 5, p. 306, May, 1968.

——— **and P. G. Neuman:** A General Purpose File System for Secondary Storage, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 213–229, 1965.

Denning, Peter J.: Effects of Scheduling on File Memory Operations, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 9–21, 1967.

———: The Working Set Model for Program Behavior, *Commun. ACM*, vol. 11, no. 5, pp. 323–333, May, 1968.

———: Thrashing: Its Causes and Prevention, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, 1968.

Dennis, J. B.: Segmentation and the Design of Multiprogrammed Computer Systems, *J. ACM*, vol. 12, no. 4, pp. 589–602, October, 1965.

——— **and E. C. Van Horn:** Programming Semantics for Multiprogrammed Computations, *Commun. ACM*, vol. 9, no. 3, pp. 143–155, March, 1966.

Dijkstra, E. W.: Structures of the Multiprogramming System, *Commun. ACM,* pp. 341–346, May, 1968.

Dodd, G. G.: Elements of Data Management Systems, *ACM Computing Surv.,* vol. 1, no. 2, pp. 113–117, June, 1969.

Estrin, G., et al.: SNUPER COMPUTER: A Computer in Instrumentation Automation, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* pp. 645–656, 1967.

Evans, D. C., and J. Y. LeClerc: Address Mapping and the Control of Access in an Interactive Computer, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* vol. 30, pp. 23–30, 1967.

Evans, G. J., Jr.: Experience Gained from the American Airline SABRE System Control Program, *Proc. ACM Natl. Conf. 1967,* pp. 77–84.

Fenichel, R. R., and A. J. Grossman: An Analytic Model of Multiprogrammed Computing, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* vol. 34, pp. 717–721, 1969.

Fikes, Richard E., et al.: Steps toward a General-purpose Time-sharing System Using Large Capacity Core Storage and TSS/360, *Proc. ACM Natl. Conf. 1968,* pp. 7–18.

Fine, G. H., et al.: Dynamic Program Behavior under Paging, *Proc. ACM Natl. Conf. 1966,* p. 223.

——— **and P. V. McIsaac:** Simulation of a Time-sharing System, *Management Sci.,* vol. 12, no. 6, pp. B180–B194, February, 1966.

Flynn, Michael J., and Donald M. MacLaren: Microprogramming Revisited, *Proc. ACM Natl. Conf. 1967,* pp. 457–463.

Freeman, David N., and Robert R. Pearson: Efficiency vs. Responsiveness in a Multiple-services Computer Facility, *Proc. ACM Natl. Conf. 1968,* pp. 25–34B.

Freibergs, I. F.: The Dynamic Behavior of Programs, *AFIPS Conf. Proc., Fall Joint Computer Conf.,* vol. 33, pt. 2, pp. 1163–1168, 1968.

Gibson, C. T.: Time-sharing in the IBM System/360: Model 67, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* vol. 28, p. 61, 1966.

Glaser, E. G., et al.: System Design of a Computer for Time-sharing Applications, *AFIPS Conf. Proc., Fall Joint Computer Conf.,* vol. 27, pp. 197–202, 1965.

Gold, M. M.: Time-sharing and Batch Processing: An Experimental Comparison of Their Values in a Problem Solving Situation, *Commun. ACM,* vol. 12, no. 5, pp. 249–259, May, 1969.

Graham, R. M.: Protection in an Information Processing Utility, *Commun. ACM,* vol. 11, no. 5, p. 365, May, 1968.

Hargraves, R. F., Jr., and A. G. Stephenson: Design Considerations for an Educational Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.,* vol. 34, pp. 657–664, 1969.

Hellerman, H.: Some Principles of Time-sharing Scheduler Strategies, *IBM Systems J.* vol. 8, no. 2, pp. 94–117, 1969.

Higgins, Alan N.: Error Recovery through Programming, *AFIPS Conf. Proc., Fall Joint Computer Conf.,* vol. 33, pt. 1, pp. 39–43, 1968.

Hobbs, W., et al.: The Baylor Medical School Teleprocessing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, p. 31, 1968.

Hoffman, L. J.: Computers and Privacy: A Survey, *ACM Computing Surv.*, vol. 1, no. 2, pp. 85–103, June, 1969.

IBM Systems J. vol. 5, no. 1, 1966 (devoted to a description of OS/360).

Jackson, P. E., and C. D. Stubbs: A Study of Multi-access Computer Communications, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 491–504, 1969.

Kay, R. H.: Management and Organization of Large Scale Software Development Projects, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 425–433, 1969.

Kilburn, T., et al.: One-Level Storage System, *IEEE Trans.*, vol. EC-11, no. 2, p. 223, 1962.

Knuth, D. E.: Fundamental Algorithms: The Art of Computer Programming, Addison-Wesley Publishing Company, Inc., Reading, Mass., 1968.

Lampson, B. W.: Scheduling and Protection on Interactive Multi-processor Systems, *Univ. Calif. Berkeley Project Genie, Doc.* 40.10.150, Jan. 20, 1967.

———: A Scheduling Philosophy for Multi-processing Systems, *Commun. ACM*, vol. 11, no. 5, pp. 347–359, May, 1968.

———: Dynamic Protection Structures, *AFIPS Conf. Proc.*, vol. 35, *Fall Joint Computer Conf.*, 1969, pp. 27–38.

———: An Overview of the CAL Timesharing System, Computer Center, Univ. of Calif., Berkeley, 1969.

——— et al.: A User Machine in a Time-sharing System, *Proc. IEEE*, December, 1966, pp. 1766–1773.

Lauer, H. C.: Bulk Core in a 360/67 Time-sharing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 601–611, 1967.

Le Clerc, Jean-Yves: Memory Structures for Interactive Computers, *Univ. Calif. Berkeley, Project Genie Doc.* 40.10.110, May, 1966.

Lichtenberger, W. W., and M. W. Pirtle: A Facility for Experimentation in Man-Machine Interaction, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, pp. 589–598, 1965.

Litt, A. S.: TSS/360: A Time-shared Operating System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 15–28, 1968.

Madnick, S. E.: A Modular Approach in File System Design, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 1–13, 1969.

Maurer, W. D.: An Improved Hash Code for Scatter Storage, *Commun. ACM*, vol. 11, no. 1, pp. 35–38, January, 1968.

McCullough, J. D., et al.: A Design for a Multiple User Multiprocessing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, p. 611, 1965.

McGeachie, J. S.: A Flexible User Validation Language for Time-sharing Systems, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 665–671, 1969.

McGee, W. C.: On Dynamic Program Relocation, *IBM Systems J.*, vol. 4, no. 3, p. 184, 1965.

McKinney, J. M.: A Survey of Analytical Time-sharing Models, *ACM Computing Surv.*, vol. 1, no. 2, pp. 105–116, June, 1969.

Merikallio, Reino A.: Simulation Design of a Multiprocessing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, 1968.

Morris, D., and F. H. Summer: An Appraisal of the Atlas Supervisor, *Proc. ACM Natl. Conf. 1967*, pp. 59–76.

Morris, Robert: Scatter Storage Techniques, *Commun. ACM*, vol. 11, no. 1, pp. 38–44, January, 1968.

Motobayashi, S., et al.: The HITAC 5020 Time Sharing System, *Proc. ACM Natl. Conf. 1969*, pp. 419–429.

Nelson, G. W.: OPTS-600: On-line Peripheral Test System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 45–50, 1968.

Newport, C. B.: Small Computers in Data Networks, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 773–775, 1969.

Nielson, N. R.: Computer Simulation of Computer System Performance, *Proc. ACM Natl. Conf. 1967*, pp. 581–590.

———: The Simulation of Time-sharing Systems, *Commun. ACM*, vol. 10, no. 7, pp. 397–412, July, 1967.

Oppenheimer, G., and N. Weizer: Resource Management for a Medium Scale Time Sharing Operating System, *Commun. ACM.*, vol. 11, no. 5, p. 313, May, 1968.

——— **and K. P. Clancy:** Considerations for Software Protection and Recovery from Hardware Failures in a Multi-access, Multiprogramming, Single Processor System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 1, pp. 29–37, 1968.

Organick, E. I.: A Guide to Multics for Subsystem Writers, *Project MAC Doc.* March, 1967.

Ossana, J. R., et al.: Communication and Input/Output Switching in a Multiplex Computing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 27, 1965.

O'Sullivan, T. C.: Exploring the Time-sharing Environment, *Proc. ACM Natl. Conf. 1967*, pp. 169–176.

Pirtle, M. W.: Intercommunication of Processors and Memory, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 621–633, 1967.

———: Modifications of the SDS-930 for the Implementation of Time-sharing, *Univ. Calif. Berkeley Project Genie Doc.* M-1, April 4, 1967.

Pyke, T. N., Jr.: Time-shared Computer Systems, *Advan. Computers*, vol. 8, 1967.

Randell, B., and C. J. Kuehner: Dynamic Storage Allocation Systems, *Commun. ACM*, vol. 11, no. 5, p. 297, May, 1968.

——— and ———: "Demand Paging in Perspective," *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1011–1018, 1968.

Rehmann, Sandra L., and Sherbie G. Gangwere, Jr.: A Simulation Study of Resource Management in a Time-sharing System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 33, pt. 2, pp. 1411–1430, 1968.

Reiter, Allen: A Resource-allocation Scheme for Multi-user On-line Operation of a Small Computer, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 30, pp. 1–7, 1967.

Rosen, Saul: Electronic Computer: A Historical Survey, *ACM Computing Surv.*, vol. 1, no. 1, pp. 7–36, March, 1969.

Rosin, Robert F.: Supervisory and Monitor Systems, *ACM Computing Surv.*, vol. 1, no. 1, pp. 37–54, March, 1969.

Sackman, H.: Time-sharing versus Batch Processing: The Experimental Evidence, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 32, pp. 1–10, 1968.

Saltzer, J. H.: Traffic Control in a Multiplexed Computer System, MAC-TR-30, thesis, Massachusetts Institute of Technology, Cambridge, Mass., July, 1966.

Schulman, Franklin D.: Hardware Measurement Devices for IBM System/360 Time-sharing Evaluation, *Proc. ACM Natl. Conf., 1967*, pp. 103–109.

Schwartz, J., E. G. Coffman, and C. Weissman: A General Purpose Time-sharing Systems, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 25, pp. 397–411, 1964.

Schwartz, J. I., and C. Weissman: The SDS Time-sharing System Revisited, *Proc. ACM Natl. Conf. 1967*, pp. 263–271.

"6700 Time-sharing Computer Reference Manual," Scientific Control Corporation, Carrollten, Texas, February, 1969 (additional information was also obtained at a public presentation of the 6700).

Shaw, J. C.: Joss: A Designer's View of an Experimental On-line Computing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 26, p. 455, 1965.

Sherr, A. L.: Analysis of Storage Performance and Dynamic Relocation Techniques, *IBM Doc.* Tr-00-1494, 1967.

———: An Analysis of Time-shared Computer Systems, Ph.D. thesis, *MIT Project MAC Doc.* MAC-TR-18, June, 1965.

Smith, Arthur A.: Input/Output in Time-shared, Segmented, Multi-processor Systems, *MIT Project MAC Doc.* (thesis) MAC-TR-28, June, 1966.

Spier, M. J., and E. I. Organick: The Multics Interprocess Communication Facility, *Second ACM Symp. on Operating Systems Principles*, Princeton Univ., October, 1969.

Sutherland, W. R., and J. W. Forgie: Graphics in Time-sharing: A Summary of the TX-2 Experience, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 620–635, 1969.

Trapnell, F. M.: A Systematic Approach to the Development of System Programs, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 411–418, 1969.

Tucker, S. G.: Microprogram Control for System/360, *IBM Systems J.*, vol. 6, no. 4, pp. 222–241, 1967.

Varian, L. C., and E. G. Coffman: An Empirical Study of the Behavior of Programs in a Paging Environment, *ACM Symp. Operation System Principles*, October, 1967.

Ver Hoef, Edward W.: Design of a Multi-level File Management System, *Proc. ACM Natl. Conf. 1966*, p. 75.

Van Horn, E. C.: Computer Design for Asynchronously Reproducible Multiprocessing, Ph.D. thesis, *MIT Project MAC Tech. Rept.* MAC-TR-34, November, 1966.

Wald, B.: The Descriptor: A Definition of the B5000 Information Processing System, Burroughs Corp., Detroit, Mich., 1961.

Watson, R. W., et al.: A Display Processor Design, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 35, 1969.

Wegner, P.: Machine Organization for Multi-programming, *Proc. ACM Natl. Conf. 1967*, pp. 135–150.

———: "Programming Languages, Information Structures and Machine Organization," McGraw-Hill Book Company, New York, 1968.

Weizer, N., and G. Oppenheimer: Virtual Memory Management in a Paging Environment, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 240–254, 1969.

Wilkes, M. V.: The Best Way to Design an Automatic Calculating Machine, *Manchester Univ. Computer Inaugural Conf., Manchester,* July, 1951, pp. 16–18.

———: Microprogramming, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 12, pp. 18–19, 1958.

———: "Time-sharing Computer Systems," American Elsevier Publishing Company, Inc., New York, 1968.

———: A Model for Core Space Allocation in a Time-sharing System, *AFIPS Conf. Proc., Spring Joint Computer Conf.*, vol. 34, pp. 265–271, 1969.

Wood, T. C.: A Generalized Supervisor for a Time-shared Operating System, *AFIPS Conf. Proc., Fall Joint Computer Conf.*, vol. 31, pp. 209–214, 1967.

Ziegler, James R.: "Time-sharing Data Processing Systems," Prentice-Hall, Inc., Englewood Cliffs, N.J., 1967.

索　　引

あ　行

か　行

さ　行

た　行

ま　行

や　行

ら　行

わ　行

<訳者略歴>

相磯秀夫（あいそ ひでお）

昭和7年　横浜に生まれる．

昭和32年　慶応義塾大学大学院工学研究科修士課程（電気工学専攻）終了．電気試験所（現，電子技術総合研究所）にてETL Mark IV の開発と工業化などに従事．通産大臣賞を受賞．

昭和46年　慶応義塾大学工学部教授．
同学部にて計算機アーキテクチャならびに計算機システム一般の研究に従事．最近はマイクロコンピュータによる複合システムなどに興味をもっている．

現　在：　慶応義塾大学理工学部電気工学科教授・工学博士．
情報処理学会理事．
情報処理学「会計算機アーキテクチャ研究会」主査．
情報処理学会，電子通信学会，電気学会，ACM，IEEE 各会員．

タイムシェアリング・システム

昭和49年11月15日　初版印刷
昭和49年12月20日　初版発行
昭和57年1月20日　2刷発行

検印省略

著者　R. W. ワトソン
訳者　相磯秀夫
発行者　村越勝弘
発行所　**日本コンピュータ協会**
東京都文京区本郷3丁目5—5
電話　東京（03）813-6929番